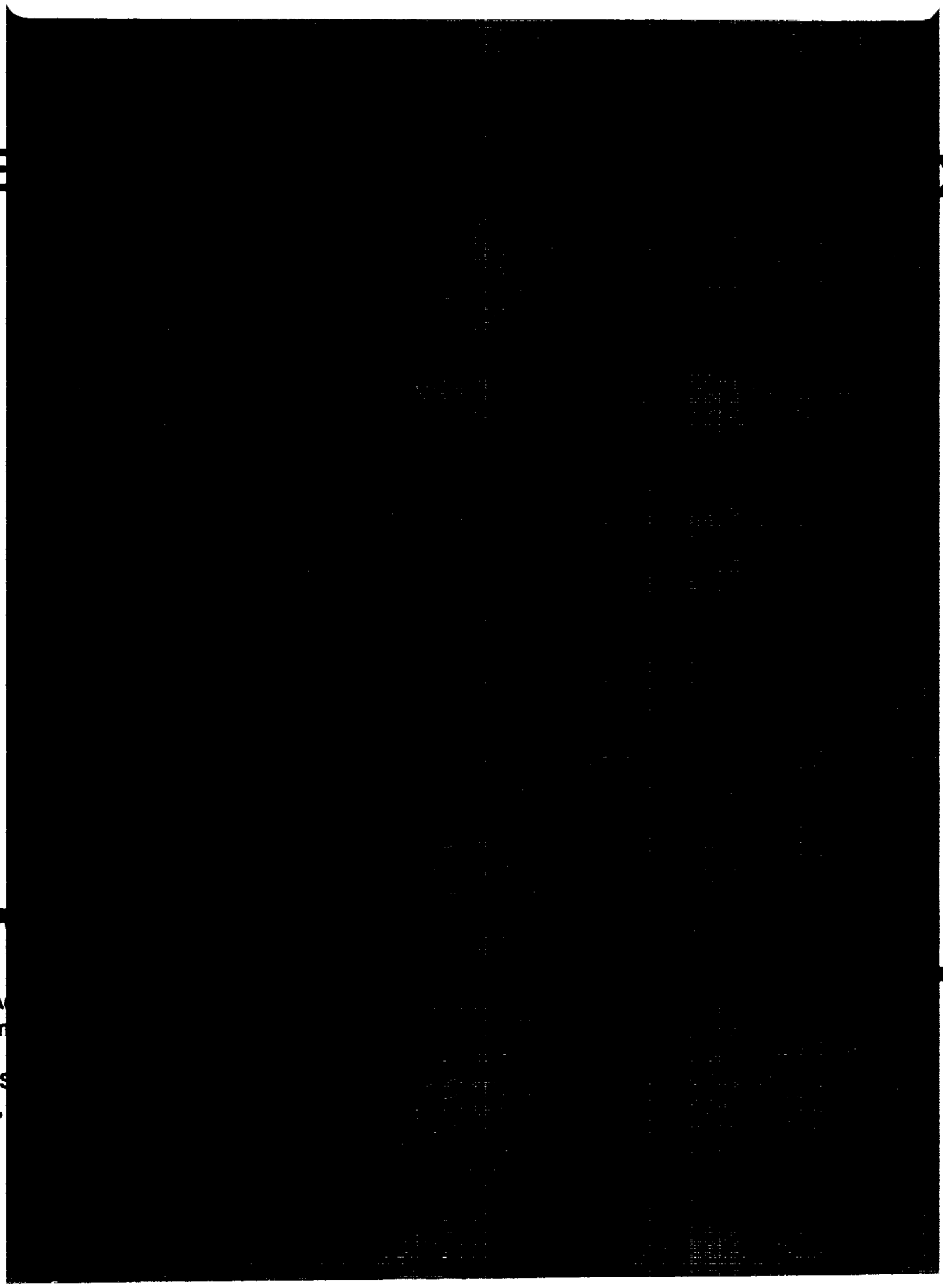
CR-189292

E X



National A
Space Adm

Goddard S
Greenbelt,

SOFTWARE ENGINEERING LABORATORY SERIES SEL-92-003

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME X

NOVEMBER 1992

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

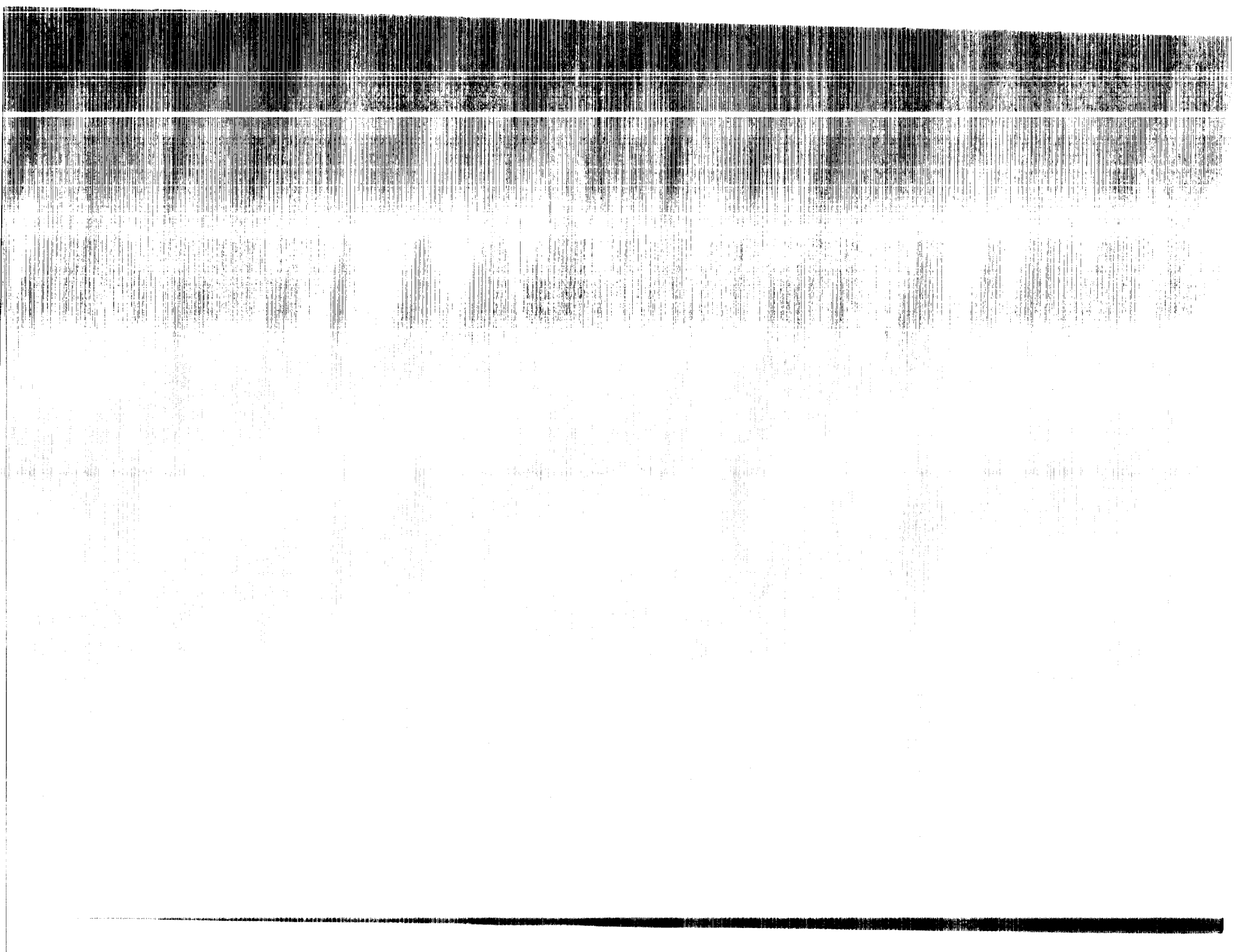
The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

TABLE OF CONTENTS

SECTION 1 – INTRODUCTION



SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from October 1991 through November 1992. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 10th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

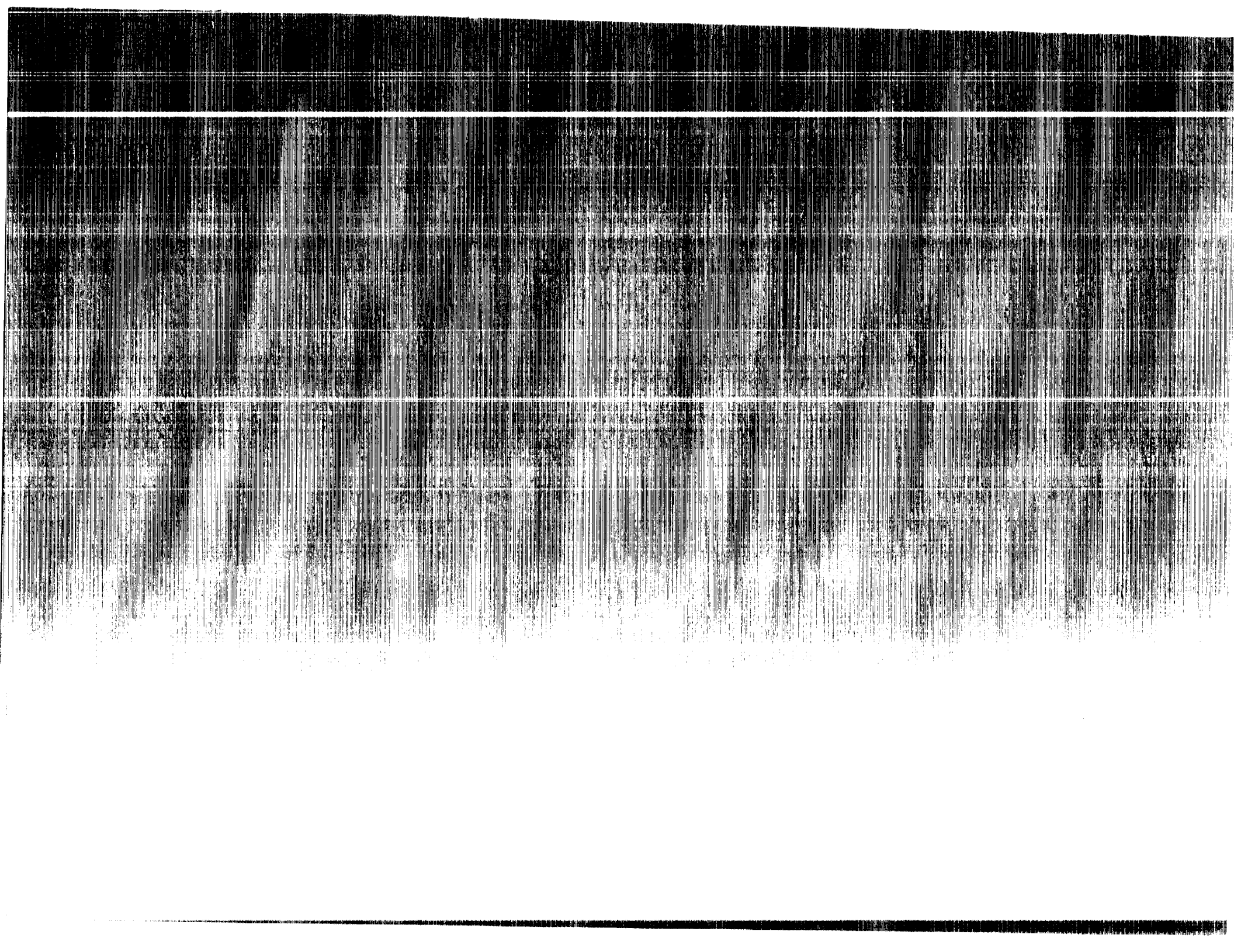
For the convenience of this presentation, the 11 papers contained here are grouped into 5 major sections:

- The Software Engineering Laboratory
- Software Tools Studies
- Software Models Studies
- Software Measurement Studies
- Ada Technology Studies

The first section (Section 2) presents a paper that characterizes the SEL as an experience factory and summarizes major lessons learned in the past 15 years. Studies on automated tools to aid in reuse and experience-based software management appear in Section 3. Section 4 includes studies on models for reuse, verification and testing phase optimization, effective management of maintenance phase changes, the software specification process, and the analysis of high-cost modules. Section 5 presents a study of maintenance measurement as it applies to the SEL. Finally, a study on the use of mixins in Ada and a summary of the performance of Ada within the SEL are included in Section 6.

The SEL is actively working to understand and improve the software development process at Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the *Collected Software Engineering Papers* and other SEL publications.

SECTION 2 – THE SOFTWARE ENGINEERING LABORATORY



SECTION 2—THE SOFTWARE ENGINEERING LABORATORY

The technical paper included in this section was originally prepared as indicated below.

- "The Software Engineering Laboratory—An Operational Software Experience Factory," V. Basili, G. Caldiera, F. McGarry, et al., *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

51-61

N93-17162 /2

THE SOFTWARE ENGINEERING LABORATORY—AN OPERATIONAL SOFTWARE EXPERIENCE FACTORY

Victor Basili and Gianluigi Caldiera
University of Maryland

Frank McGarry and Rose Pajerski
National Aeronautics and Space Administration/
Goddard Space Flight Center

Gerald Page and Sharon Waligora
Computer Sciences Corporation

ABSTRACT

For 15 years, the Software Engineering Laboratory (SEL) has been carrying out studies and experiments for the purpose of understanding, assessing, and improving software and software processes within a production software development environment at the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). The SEL comprises three major organizations:

- NASA/GSFC, Flight Dynamics Division
- University of Maryland, Department of Computer Science
- Computer Sciences Corporation, Flight Dynamics Technology Group

These organizations have jointly carried out several hundred software studies, producing hundreds of reports, papers, and documents, all of which describe some aspect of the software engineering technology that has been analyzed in the flight dynamics environment at NASA. The studies range from small, controlled experiments (such as analyzing the effectiveness of code reading versus that of functional testing) to large, multiple-project studies (such as assessing the impacts of Ada on a production environment). The organization's driving goal is to improve the software process continually, so that sustained improvement may be observed in the resulting products. This paper discusses the SEL as a functioning example of an operational software experience factory and summarizes the characteristics of and major lessons learned from 15 years of SEL operations.

1. THE EXPERIENCE FACTORY CONCEPT

Software engineering has produced a fair amount of research and technology transfer in the first 24 years of its existence. People have built technologies, methods, and tools that are used by many organizations in development and maintenance of software systems.

Unlike other disciplines, however, very little research has been done in the development of models for the various components of the discipline. Models have been developed primarily for the software product, providing mathematical models of its function and structure (e.g., finite state machines in object-oriented design), or, in some advanced instances, of its observable quality (e.g., reliability models). However, there has been very little modeling of several other important components of the software engineering discipline, such as processes, resources, and defects. Nor has much been done toward understanding the logical and physical integration of software engineering models, analyzing and evaluating them via experimentation and simulation, and refining and tailoring them to the characteristics and the needs of a specific application environment.

Currently, research and technology transfer in software engineering are done mostly bottom-up and in isolation. To provide software engineering with a rigorous, scientific foundation and a pragmatic framework, the following are needed [1]:

- A top-down, experimental, evolutionary framework in which research can be focused and logically integrated to produce models of the discipline, which can then be evaluated and tailored to the application environment
- An experimental laboratory associated with the software artifact that is being produced and studied to develop and refine comprehensive models based upon measurement and evaluation

The three major concepts supporting this vision are

- A concept of evolution: the Quality Improvement Paradigm [2]
- A concept of measurement and control: the Goal/Question/Metric Approach [3]
- A concept of the organization: the Experience Factory [4]

The **Quality Improvement Paradigm** is a two-feedback loop process (project and organization loops) that is a variation of the scientific method. It consists of the following steps:

- Characterization: Understand the environment based upon available models, data, intuition, etc., so that similarities to other projects can be recognized
- Planning: Based on this characterization:
  - Set quantifiable goals for successful project and organization performance and improvement
  - Choose the appropriate processes for improvement, and supporting methods and tools to achieve the goals in the given environment
- Execution: Perform the processes while constructing the products and provide real-time project feedback based on the goal achievement data
- Packaging: At the end of each specific project:
  - Analyze the data and the information gathered to evaluate the current practices, determine problems,

record findings, and make recommendations for future project improvements

- Package the experience gained in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects

- Store the packages in an experience base so they are available for future projects

The **Goal/Question/Metric Approach** is used to define measurement on the software project, process, and product in such a way that

- Resulting metrics are tailored to the organization and its goals
- Resulting measurement data play a constructive and instructive role in the organization
- Metrics and their interpretation reflect the quality values and the different viewpoints (developers, users, operators, etc.)

Although originally used to define and evaluate a particular project in a particular environment, the Goal/Question/Metric Approach can be used for control and improvement of a software project in the context of several projects within the organization [5,6].

The Goal/Question/Metric Approach defines a measurement model on three levels:

- Conceptual level (goal): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, and relative to a particular environment
- Operational level (question): A set of questions is used to define models of the object of study and the focuses on that object to characterize the assessment or achievement of a specific goal
- Quantitative level (metric): A set of metrics, based on the models, is associated with every question in order to answer it in a quantitative way

The concept of the **Experience Factory** was introduced to institutionalize the collective learning of the organization that is at the root of continual improvement and competitive advantage.

Reuse of experience and collective learning cannot be left to the imagination of individual, very talented, managers: they become a corporate concern, like the portfolio of a business or company assets. The experience factory is the organization that supports reuse of experience and collective learning by developing, updating, and delivering, upon request to the project organizations, clusters of competencies that the SEL refers to as experience packages. The project organizations offer to the experience factory their products, the plans used in their development, and the data gathered during development and operation (Figure 1). The experience factory transforms these objects into reusable units and supplies them to the project organizations, together with specific support that includes monitoring and consulting (Figure 2).

The experience factory can be a logical and/or physical organization, but it is important that its activities are separated and made independent from those of the project organization. The packaging of

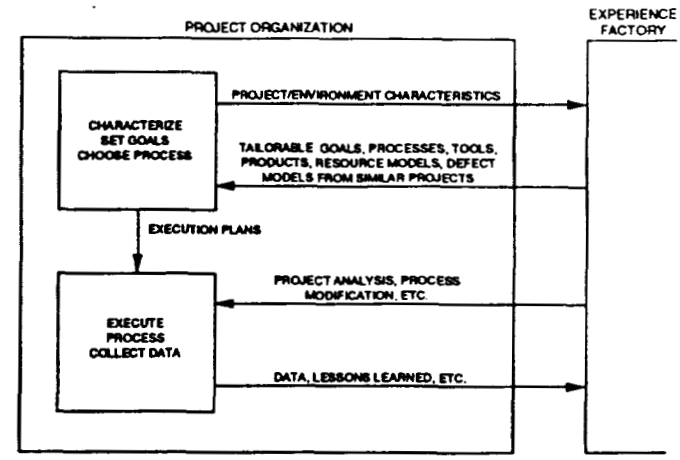


Figure 1. Project Organization Functions

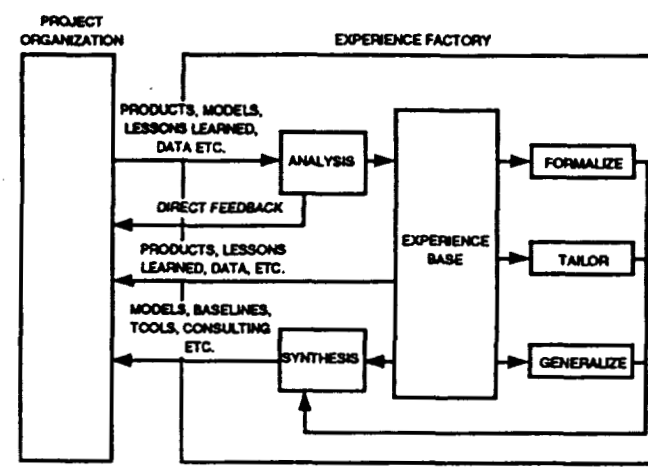


Figure 2. Experience Factory Functions

experience is based on tenets and techniques that are different from th problem solving activity used in project development [7].

On the one hand, from the perspective of an organization producir software, the difference is outlined in the following chart:

PROJECT ORGANIZATION (Problem Solving)	EXPERIENCE FACTORY (Experience Packaging)
Decomposition of a problem into simpler ones	Unification of different solutions and redefinition of the problem
Instantiation	Generalization, formalization
Design/implementation process	Analysis/synthesis process
Validation and verification	Experimentation

On the other hand, from the perspective of software engineering research, there are the following goals:

PROJECT ORGANIZATION (Problem Solving)	EXPERIENCE FACTORY (Experience Packaging)
Develop representative languages for products processes	Develop techniques for abstraction generalization tailoring formalization analysis/synthesis
Develop techniques for design/implementation data collection/validation/analysis validation and verification	Experiment with techniques
Build automatic support tools	Package and integrate for reuse experimental results processes/products

In a correct implementation of the experience factory paradigm, the projects and the factory will have different process models. Each project will choose its process model based on the characteristics of the software product that will be delivered, whereas the factory will define (and change) its process model based upon organizational and performance issues. The main product of the experience factory is the experience package. There are a variety of software engineering experiences that can be packaged: resource baselines and models; change and defect baselines and models; product baselines and models; process definitions and models; method and technique models and evaluations; products; lessons learned; and quality models. The content and structure of an experience package vary based on the kind of experience clustered in the package. There is, generally, a central element that determines what the package is: a software life-cycle product or process, a mathematical relationship, an empirical or theoretical model, a data base, etc. This central element can be used to identify the experience package and produce a taxonomy of experience packages based on the characteristics of this central element:

- Product packages (programs, architectures, designs)
- Tool packages (constructive and analytic tools)
- Process packages (process models, methods)
- Relationship packages (cost and defect models, resource models, etc.)
- Management packages (guidelines, decision support models)
- Data packages (defined and validated data, standardized data, etc.)

The structure and functions of an efficient implementation of the experience factory concept are modeled on the characteristics and the goals of the organization it supports. Therefore, different levels of abstraction best describe the architecture of an experience factory in order to introduce the specificity of each environment at the right level without losing the representation of the global picture and the ability to compare different solutions [8].

The levels of abstraction that the SEL proposes to represent the architecture of an experience factory are as follows:

- Reference level: This first and more abstract level represents the activities in the experience factory by active objects, called architectural agents. They are specified by their ability to perform specific tasks and to interact with each other.
- Conceptual level: This level represents the interface of the architectural agents and the flows of data and control among them. They specify who communicates with whom, what is done in the experience factory, and what is done in the project organization. The boundary of the experience factory, i.e., the line that separates it from the project organization, is defined at this level based on the needs and characteristics of an organization. It can evolve as these needs and characteristics evolve.
- Implementation level: This level defines the actual technical and organizational implementation of the architectural agents and their connections at the conceptual level. They are assigned process and product models, synchronization and communication rules, and appropriate performers (people or computers). Other implementation details, such as mapping the agents over organizational departments, are included in the specifications provided at this level.

The architecture of the experience factory can be regarded as a special instance of an experience package whose design and evolution are based on the levels of abstraction just introduced and on the methodological framework of the improvement paradigm applied to the specific architecture.

The Software Engineering Laboratory (SEL) is an operating example of an experience factory. Figure 3 shows the conceptual level of the SEL experience factory, identifying the primary architectural agents and the interactions among them. The remaining sections describe the SEL implementation of the experience factory concept. They discuss its background, operations, and achievements, and assess the impact it has had on the production environment it supports.

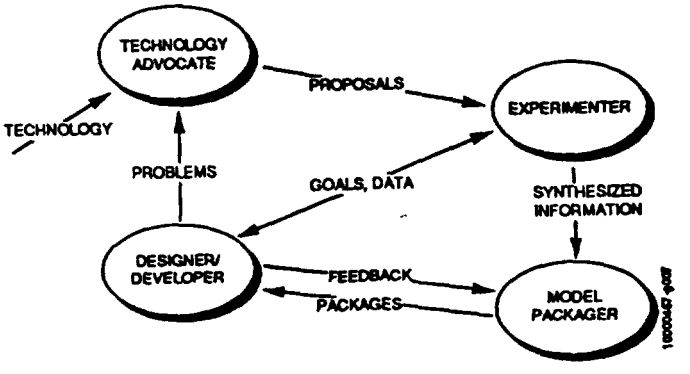


Figure 3. The SEL—Conceptual Level

2. SEL BACKGROUND

The SEL was established in 1976 as a cooperative effort among the University of Maryland, the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC), and Computer Sciences Corporation (CSC). Its goal was to understand and improve the software development process and its products within GSFC's Flight Dynamics Division (FDD). At that time, although significant advances were being made in developing new technologies (e.g., structured development practices, automated tools, quality assurance approaches, and management tools), there was very limited empirical evidence or guidance for applying these promising, yet immature, techniques. Additionally, it was apparent that there was very limited evidence available to qualify or to

quantify the existing software process and associated products, let alone understand the impact of specific process methods. Thus, the SEL staff initiated efforts to develop some means by which the software process could be understood (through measurement), qualified, and measurably improved through continually expanding understanding, experimentation, and process refinement.

This working relationship has been maintained continually since its inception with relatively little change to the overall goals of the organization. In general, these goals have matured rather than changed; they are as follows:

1. Understand: Improve insight into the software process and its products by characterizing a production environment.
2. Assess: Measure the impact that available technologies have on the software process. Determine which technologies are beneficial to the environment and, most importantly, how the technologies must be refined to best match the process with the environment.
3. Package/Infuse: After identifying process improvements, package the technology in a form that allows it to be applied in the production organization.

These goals are addressed sequentially, in an iterative fashion, as shown in Figure 4.

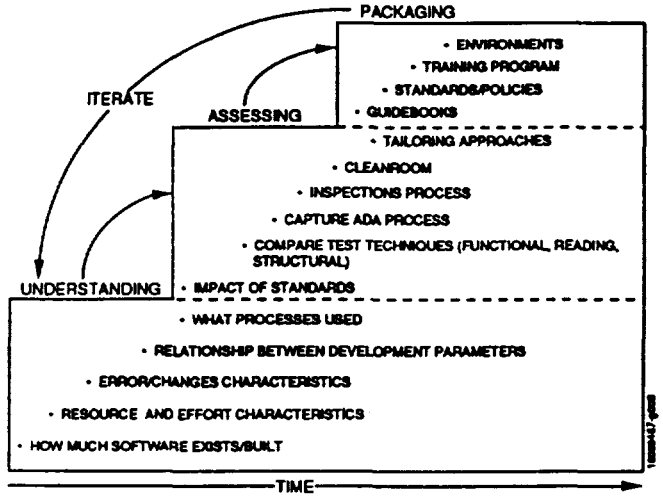


Figure 4. SEL Process Improvement Steps

The approach taken to attaining these goals has been to apply potentially beneficial techniques to the development of production software and to measure the process and product in enough detail to quantifiably assess the applied technology. Measures of concern, such as cost, reliability, and/or maintainability, are defined as the organization determines the major near- and long-term objectives for its software development process improvement program. Once those objectives are known, the SEL staff designs the experiment; that is, it defines the particular data to be captured and the questions that must be addressed in each experimental project.

All of the experiments conducted by the SEL have occurred within the production environment of the flight dynamics software development facility at NASA/GSFC. The SEL production environment consists of projects that are classified as mid-sized software systems. The average project lasts 2 to 3-1/2 years, with an average staff size of 15 software developers. The average software size is 175 thousand source lines of code (KSLOC), counting commentary, with about 25 percent reused from previous development efforts. Virtually all projects in this environment are scientific ground-based systems, although some embedded systems have been developed. Most software is developed in FORTRAN, although Ada is starting to be used more frequently. Other languages, such as Pascal and Assembly, are used occasionally. Since this environment is relatively consistent, it is conducive to the experimentation process. In the SEL, there exists a homogeneous class of software, a stable development environment, and a controlled, consistent, management and development process.

3. SEL OPERATIONS

The following three major functional groups support the experimentation and studies within the SEL (Figure 5):

1. Software developers, who are responsible for producing the flight dynamics application software
2. Software engineering analysts, who are the researchers responsible for carrying out the experimentation process and producing study results
3. Data base support staff, who are responsible for collecting, checking, and archiving all of the information collected from the development efforts

During the past 15 years, the SEL has collected and archived data on over 100 software development projects in the organization. The data are also used to build typical project profiles against which ongoing projects can be compared and evaluated. The SEL provides managers in this environment with tools (online and paper) for monitoring and assessing project status.

Typically, there are 6 to 10 projects simultaneously in progress in the flight dynamics environment. As was mentioned earlier, they average 175 KSLOC, ranging from small (6–8 KSLOC) to large (300–400 KSLOC), with a few exceeding 1 million source lines of code (MSLOC). Each project is considered an experiment within the SEL, and the goal is to extract detailed information to understand the process better and to provide guidance to future projects.

To support the studies and to support the goal of continually increasing understanding of the software development process, the SEL regularly collects detailed data from its development projects. The types of data collected include cost (measured in effort), process, and product data. Process data include information about the project, such as the methodology, tools and techniques used, and information about personnel experience and training. Product data include size (in SLOC), change and error information, and the results of postdevelopment static analysis of the delivered code.

The data may be somewhat different from one project to another since the goals for a particular experiment may be different between projects. There is a basic set of information (such as effort and error data) that is collected for every project. However, as changes are made to specific processes (e.g., Ada projects), the detailed data collected may be modified. For example, Figure 6 shows the standard error report form, used on all projects, and the modified Ada version, used for specific projects where Ada is being studied.

As the information is collected, it is quality assured and placed in a central data base. The analysts then use these data together with other information, such as subjective lessons learned, to analyze the impact of a specific software process and to measure and then feedback results to both ongoing projects and follow-on projects.

The data are used to build predictive models for future projects and to provide a rationale for refining particular software processes being used. As the data are analyzed, papers and reports are generated that reflect results of the numerous studies. Additionally, the results of the analysis are packaged as standards, policies, training materials, and management tools.

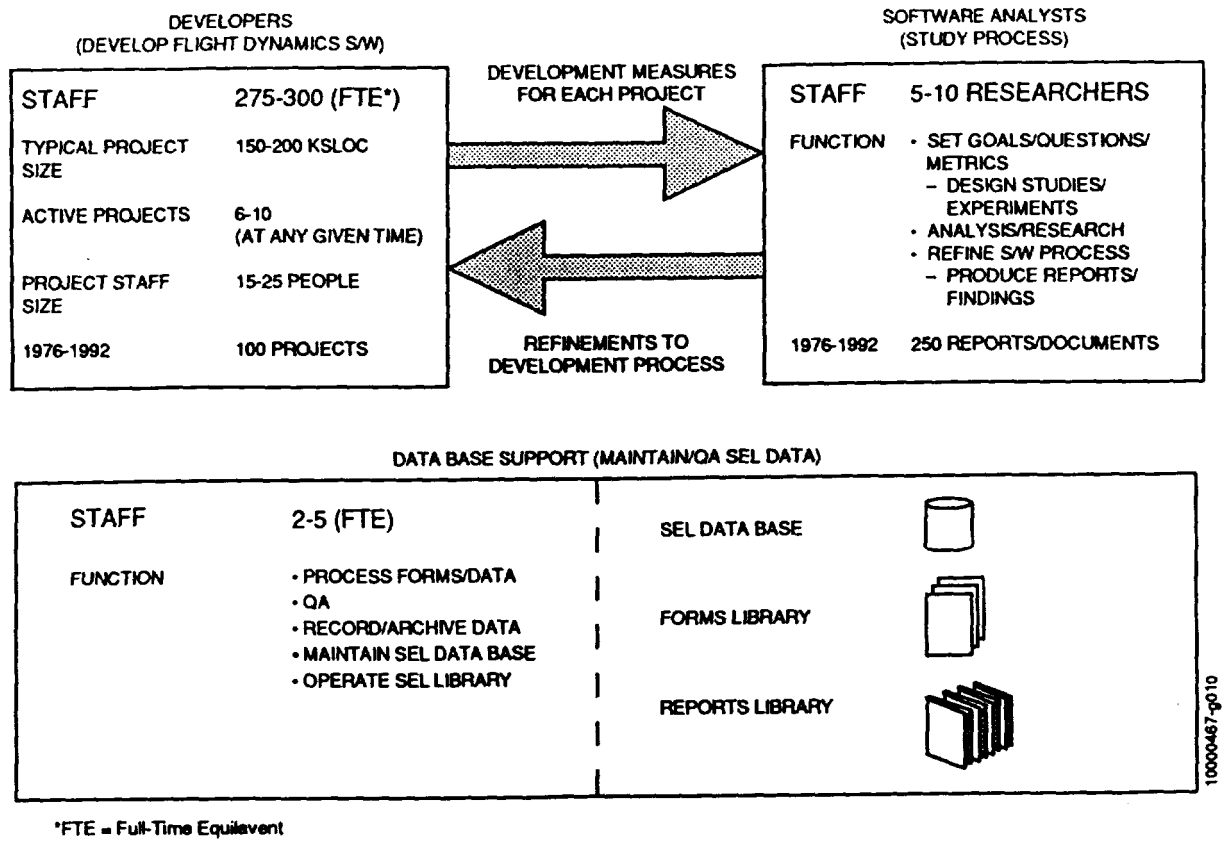


Figure 5. SEL Functional Groups

4. SEL DATA ANALYSIS

The overall concept of the experience factory has continually matured within the SEL as understanding of the software process has increased. The experience factory goal is to demonstrate continual improvement of the software process within an environment by carrying out analysis, measurement, and feedback to projects within the environment. The steps, previously described, include **understanding, assessment/refinement, and packaging.** The data described in the previous section are used as one major element that supports these three activities in the SEL. In this section, examples are given to demonstrate the major stages of the experience factory.

4.1. UNDERSTANDING

Understanding what an organization does and how that organization operates is fundamental to any attempt to plan, manage, or improve the software process. This is especially true for software development organizations. The following two examples illustrate how understanding is supported in an operation such as the SEL.

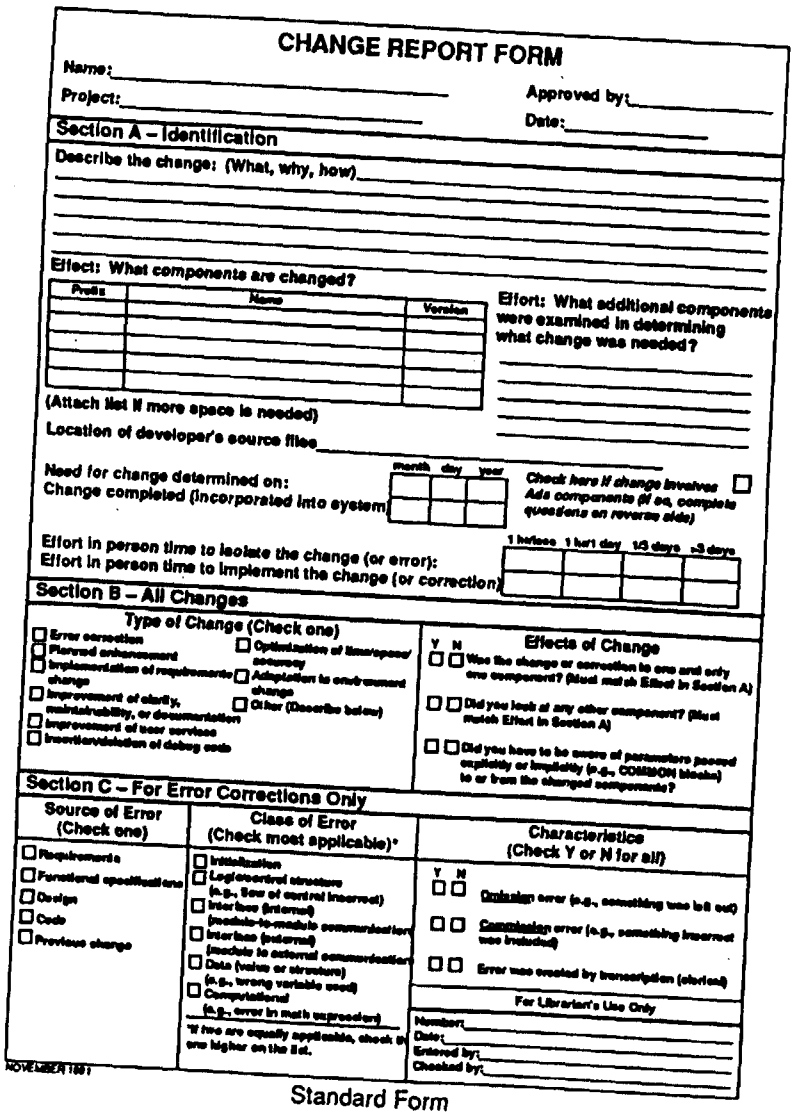
Effort distribution (i.e., which phases of the life cycle consume what portion of development effort) is one baseline characteristic of the SEL software development process. Figure 7 presents the effort distributions for 11 FORTRAN projects, by life-cycle phase and by activity. The phase data count hours charged to a project during each calendar phase. The activity data count all hours attributed to a particular activity (as reported by the programmer), regardless of when in the life cycle the activity occurred. Understanding these distributions is important to assessing the similarities/differences observed on an ongoing project, planning new efforts, and evaluating new technology.

The error detection rate is another interesting model from the SEL environment. There are two types of information in this model. The first is the absolute error rate expected in each phase. By collecting the information on software errors, the SEL has constructed a model of the expected error rate in each phase of the life cycle. The SEL expects about four errors per 1000 SLOC during implementation: two during system test, one during acceptance test, and one-half during operation and maintenance. Analysis of more recent projects indicates that these absolute error rates are declining as the software development process and technology improve.

The trend that can be derived from this model is that the error detection rates reduce by 50 percent in each subsequent phase (Figure 8). This pattern seems to be independent of the actual values of the error rates; it is still true in the recent projects where the overall error rates are declining. This model of error rates, as well as numerous other similar types of models, can be used to better predict, manage, and assess change on newly developed projects.

4.2. ASSESSING/REFINING

In the second major stage of the experience factory, elements of the process (such as specific software development techniques) are assessed, and the evolving technologies are tailored to the particular environment. Each project in the SEL is considered to be an experiment in which some software method is studied in detail. Generally, the subject of the study is a specific modification to the standard process, a process that obviously comprises numerous software methods.



CHANGE REPORT FORM

Name: _____ Approved by: _____

Project: _____ Date: _____

Section A – Identification

Describe the change: (What, why, how) _____

Effect: What components are changed?

Prefix	Name	Version

(Attach list if more space is needed)

Location of developer's source files _____

Effort: What additional components were examined in determining what change was needed? _____

Need for change determined on: month day year

Change completed (incorporated into system) month day year

Check here if change involves Ada components (if so, complete questions on reverse side) ☐

Effort in person time to isolate the change (or error): 1 hr/less | 1 hr/1 day | 1/3 days | >3 days

Effort in person time to implement the change (or correction): 1 hr/less | 1 hr/1 day | 1/3 days | >3 days

Section B – All Changes

Type of Change (Check one)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Describe below)

Effects of Change

Y	N	
☐	☐	Was the change or correction to one and only one component? (Must match Effect in Section A)
☐	☐	Did you look at any other component? (Must match Effect in Section A)
☐	☐	Did you have to be aware of parameters passed explicitly or implicitly (e.g., COMMON blocks) to or from the changed components?

Section C – For Error Corrections Only

Source of Error (Check one)

- ☐ Requirements
- ☐ Functional specifications
- ☐ Design
- ☐ Code
- ☐ Previous change

Class of Error (Check most applicable)*

- ☐ Initialization
- ☐ Logic/control structure (e.g., flow of control incorrect)
- ☐ Interface (internal) (module-to-module communication)
- ☐ Interface (external) (module to external communication)
- ☐ Data (value or structure) (e.g., wrong variable used)
- ☐ Computational (e.g., error in math expression)

*If two are equally applicable, check the one higher on the list.

Characteristics (Check Y or N for all)

Y	N	
☐	☐	Omission error (e.g., something was left out)
☐	☐	Commission error (e.g., something incorrect was included)
☐	☐	Error was created by transcription (clerical)

For Librarian's Use Only

Number: _____

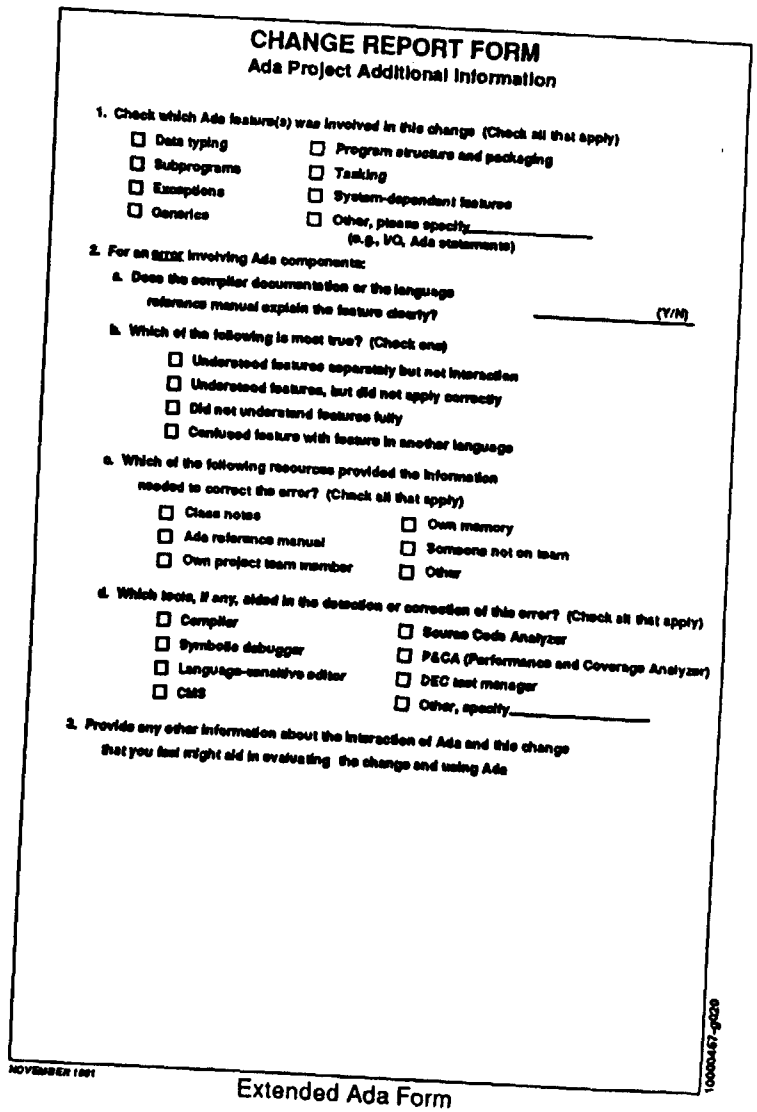
Date: _____

Entered by: _____

Checked by: _____

NOVEMBER 1981

Standard Form



CHANGE REPORT FORM

Ada Project Additional Information

1. Check which Ada feature(s) was involved in this change (Check all that apply)
   - ☐ Data typing
   - ☐ Subprograms
   - ☐ Exceptions
   - ☐ Generics
   - ☐ Program structure and packaging
   - ☐ Tasking
   - ☐ System-dependent features
   - ☐ Other, please specify _____ (e.g., I/O, Ada statements)
2. For an error involving Ada components:
   a. Does the compiler documentation or the language reference manual explain the feature clearly? _____ (Y/N)
   b. Which of the following is most true? (Check one)
      - ☐ Understood features separately but not interaction
      - ☐ Understood features, but did not apply correctly
      - ☐ Did not understand features fully
      - ☐ Confused feature with feature in another language
   c. Which of the following resources provided the information needed to correct the error? (Check all that apply)
      - ☐ Class notes
      - ☐ Ada reference manual
      - ☐ Own project team member
      - ☐ Own memory
      - ☐ Someone not on team
      - ☐ Other
   d. Which tools, if any, aided in the detection or correction of this error? (Check all that apply)
      - ☐ Compiler
      - ☐ Symbolic debugger
      - ☐ Language-sensitive editor
      - ☐ CMS
      - ☐ Source Code Analyzer
      - ☐ P&CA (Performance and Coverage Analyzer)
      - ☐ DEC test manager
      - ☐ Other, specify _____
3. Provide any other information about the interaction of Ada and this change that you feel might aid in evaluating the change and using Ada

NOVEMBER 1981

10000467-gf029

Extended Ada Form

Figure 6. Error Report Forms

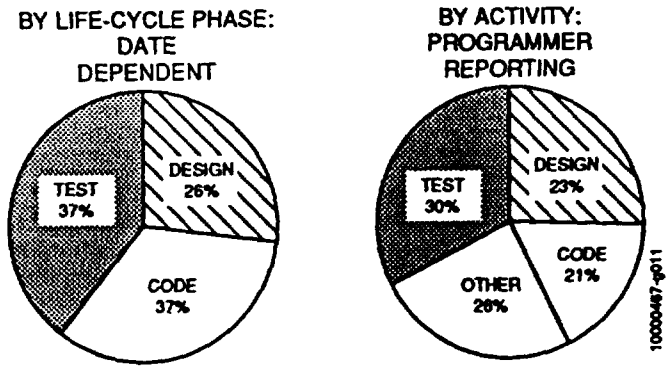


Figure 7. Effort Distribution

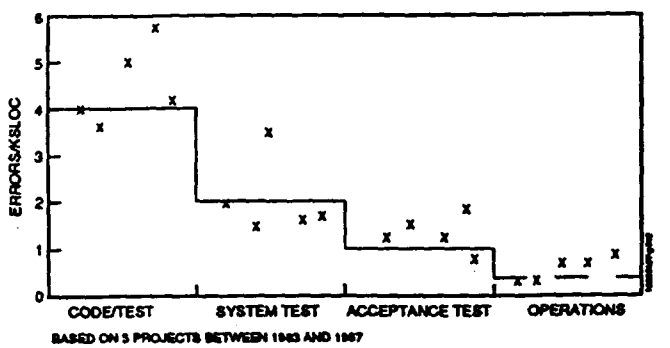


Figure 8. Derived SEL Error Model

One recent study that exemplifies the assessment stage involves the Cleanroom software methodology [9]. This methodology has been applied on three projects within the SEL, each providing additional insight into the Cleanroom process and each adding some element of "refinement" to the methodology for this one environment.

The SEL trained teams in the methodology, then defined a modified set of Cleanroom-specific data to be collected. The projects were studied in an attempt to assess the impact that Cleanroom had on the process as well as on such measures as productivity and reliability. Figure 9 depicts the characteristics of the Cleanroom changes, as well as the results of the three experiments.

The Cleanroom experiments included significant changes to the standard SEL development methodology, thereby requiring extensive training, preparation, and careful execution of the studies. Detailed experimentation plans were generated for each of the studies (as they are for all such experiments), and each included a description of the goals, the questions that had to be addressed, and the metrics that had to be collected to answer the questions.

Since this methodology consists of multiple specific methods (e.g., box structure design, statistical testing, rigorous inspections), each particular method had to be analyzed along with the full, integrated, Cleanroom methodology in general. As a result of the analysis, Cleanroom has been "assessed" as a beneficial approach for the SEL (as measured by specific goals of these studies), but specific elements of the full methodology had to be tailored to better fit the particular SEL environment. The tailoring and modifying resulted in a revised Cleanroom process model, written in the form of a process handbook [10], for future applications to SEL projects. That step is the "packaging" component of the experience factory process.

4.3. PACKAGING

The final stage of a complete experience factory is that of packaging. After beneficial methods and technologies are identified, the organization must provide feedback to ensuing projects by capturing the process in the form of standards, tools, and training. The SEL has produced a set of standards for its own use that reflect the results of the studies it has conducted. It is apparent that such standards must continually evolve to capture modified characteristics of the process. (The SEL typically updates its basic standard every 5 years.) Examples of standards that have been produced as part of the packaging process include:

- *Manager's Handbook for Software Development [11]*
- *Recommended Approach to Software Development [12]*

One additional example of an extensive packaging effort in the SEL is a management tool called the Software Management Environment (SME). The concepts of the SME, which is now an operational tool used locally in the SEL, have evolved over 8 years. This tool accesses SEL project data, models, relationships, lessons learned, and managers' rules of thumb to present project characteristics to the manager of an ongoing project. This allows the manager to gain insight into the project's consistency with or deviation from the norm for the environment (Figure 10).

This example of "packaging" reflects the emphasis that must be placed on making results of software projects, in the form of lessons learned, refined models, and general understanding, easily available to other follow-on development projects in a particular organization.

The tool searches the collection of 15 years of experience archived in the SEL to select appropriate, similar project data so that managers can plan, monitor, predict, and better understand their own project based on the analyzed history of similar software efforts.

As an example, all of the error characteristics of the flight dynamics projects have resulted in the error model depicted in Figure 8, where history has shown typical software error rates in the different phases of the life cycle. As new projects are developed and error discrepancies are routinely reported and added to the SEL data base, the manager can easily compare error rates on his or her project with typical error rates on completed, similar projects. Obviously, the data are environment dependent, but the concepts of measurement, process improvement, and packaging are applicable to all environments.

5. ADA ANALYSIS

A more detailed example of one technology that has been studied in the SEL within the context of the experience factory is that of Ada. By 1985, the SEL had achieved a good understanding of how software was developed in the FDD; it had baselined the development process and had established rules, relationships, and models that improved the manageability of the process. It had also fine-tuned its process by adding and refining techniques within its standard methodology. Realizing that Ada and object-oriented techniques offered potential for major improvement in the flight dynamics environment, the SEL decided to pursue experimentation with Ada.

The first step was to set up expectations and goals against which results would be measured. The SEL's well-established baseline and set of measures provided an excellent basis for comparison. Expectations included a change in the effort distribution of development activities (e.g., increased design and decreased testing); no greater cost per new line of code; increased reuse; decreased maintenance costs; and increased reliability (i.e., lower error rates, fewer interface errors, and fewer design errors).

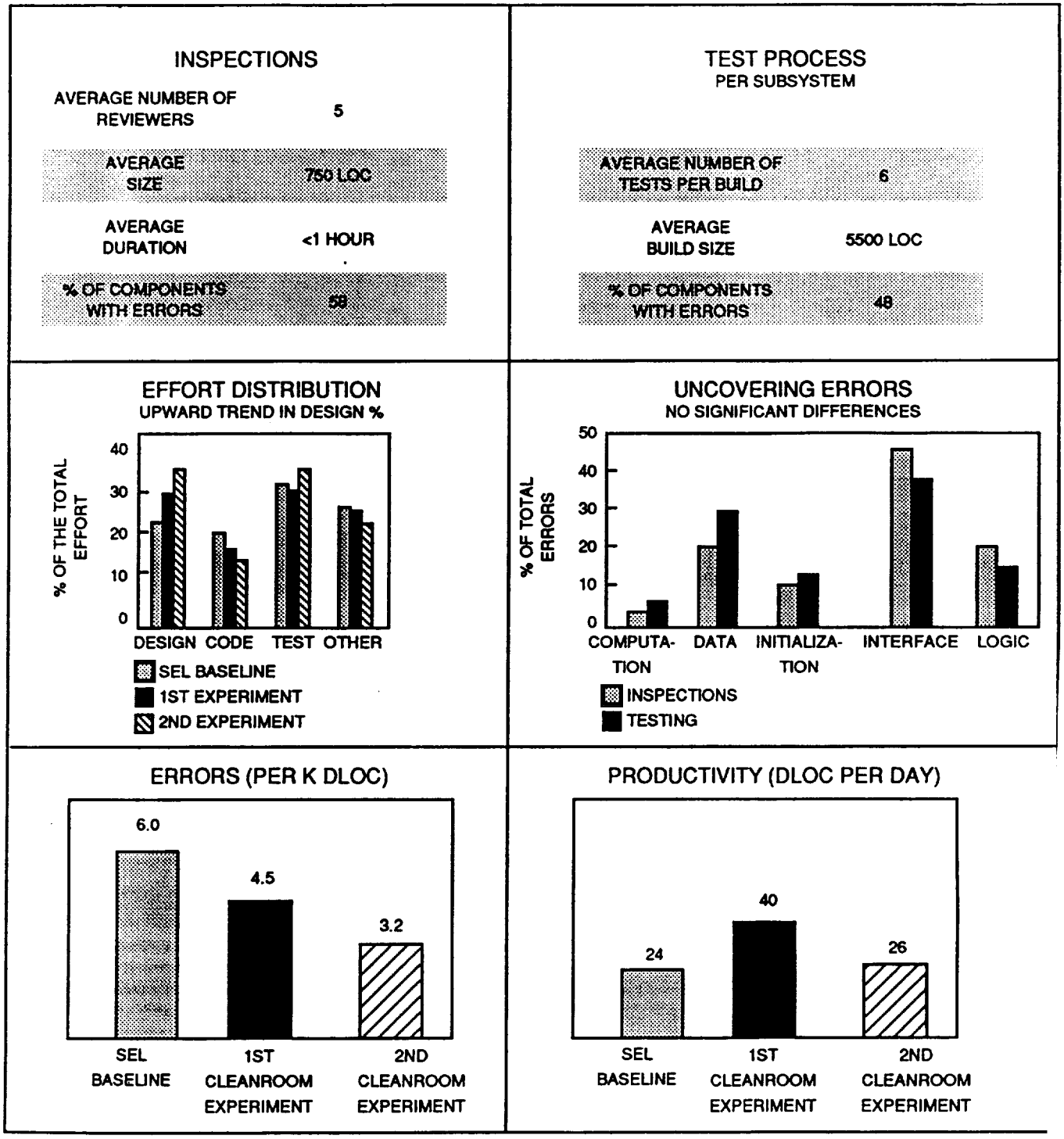


Figure 9. Cleanroom Assessment in the SEL

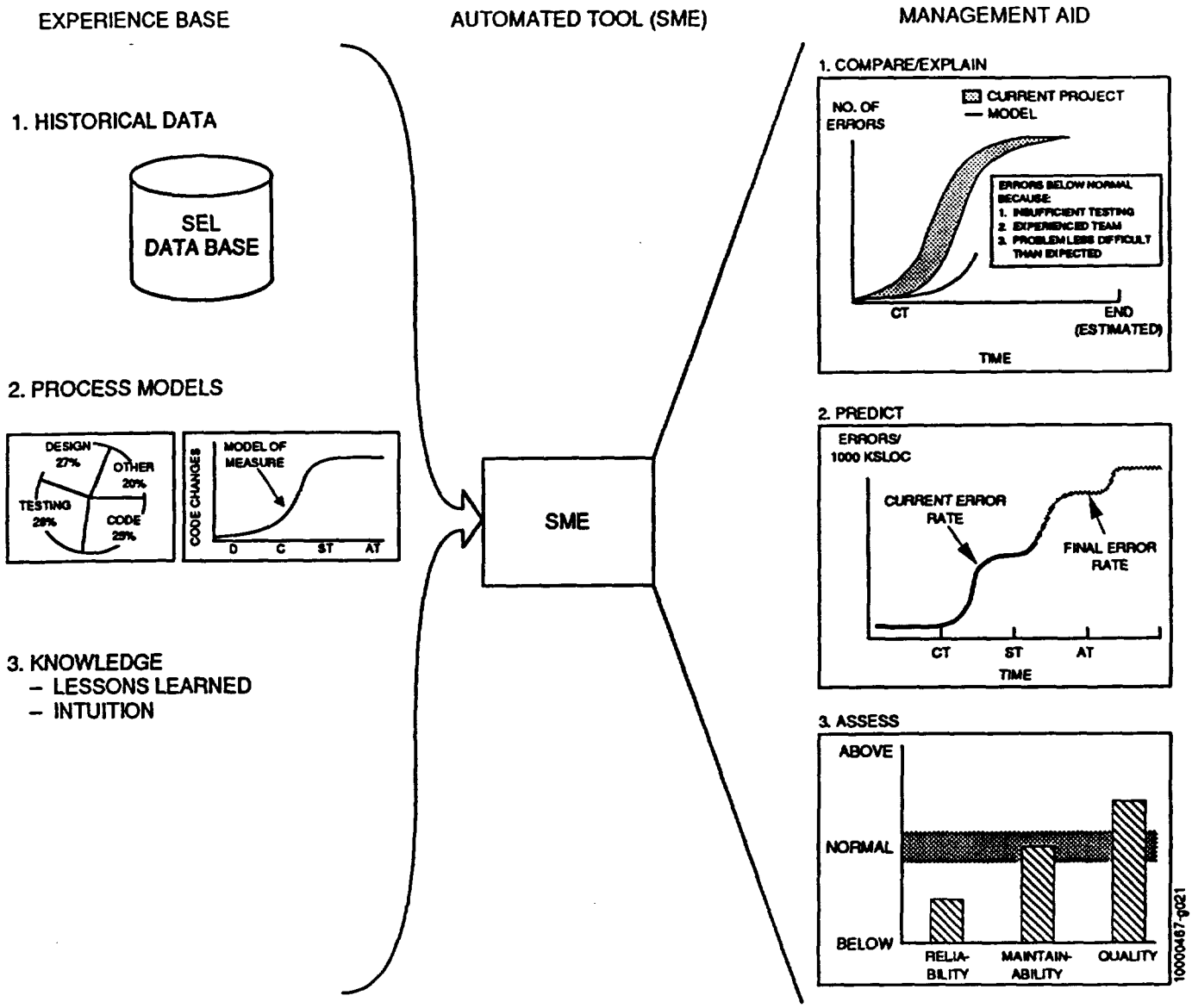


Figure 10. SME: A Tool for "Packaging"

The SEL started with a small, controlled experiment in which two versions of the same system were developed in parallel: one was developed in FORTRAN using the standard SEL structured methodology, and the other was developed in Ada using an object-oriented development (OOD) methodology. Because the Ada system would not become operational, analysts had time to investigate new ideas and learn about the new technology while extracting good calibration information for comparing FORTRAN and Ada projects, such as size ratios, average component size, error rates, and productivity. These data provided a reasonable means for planning the next set of Ada projects that, even though they were small, would deliver mission support software.

Over the past 6 years the SEL has completed 10 Ada/OOD projects, ranging in size from 38 to 185 KSLOC. As projects completed and new ones started, the methodology was continually evaluated and refined. Some characteristics of the Ada environment emerged early and have remained rather constant; others took time to stabilize. For example, Ada projects have shown no significant change in effort distribution or in error classification when compared with the SEL FORTRAN baseline. However, reuse has increased dramatically, as shown in Figure 11.

Over the 6-year period, the use of Ada and OOD has matured. Source code analysis of the Ada systems, grouped chronologically, revealed a maturing use of key Ada features, such as generics, strong typing, and packaging, whereas other features, such as tasking, were deemed inappropriate for the application. Generics, for example, were not only used more often in the recent systems, increasing from 8 to 50 percent of the system, but they were also used in more sophisticated ways, so that parameterization increased eightfold. Moreover, the use of Ada features has stabilized over the last 3 years, creating a SEL baseline for Ada development.

The cost to develop new Ada code has remained higher than the cost to develop new FORTRAN code. However, because of the high reuse, the cost to deliver an Ada system has significantly

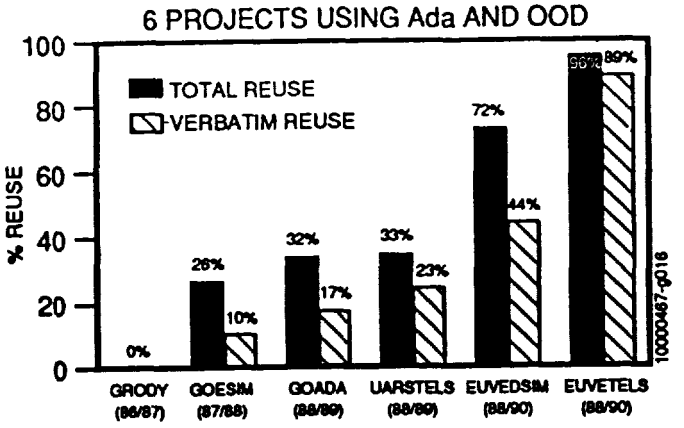


Figure 11. Reuse Trends

decreased and is now well below the cost to deliver an equivalent FORTRAN system (Figure 12).

Reliability of Ada systems has also improved as the environment has matured. Although the error rates for Ada systems, shown in Figure 13, were significantly lower from the start than those for FORTRAN, they have continued to decrease even further. Again, the high level of reuse in the later systems is a major contributor to this greatly improved reliability.

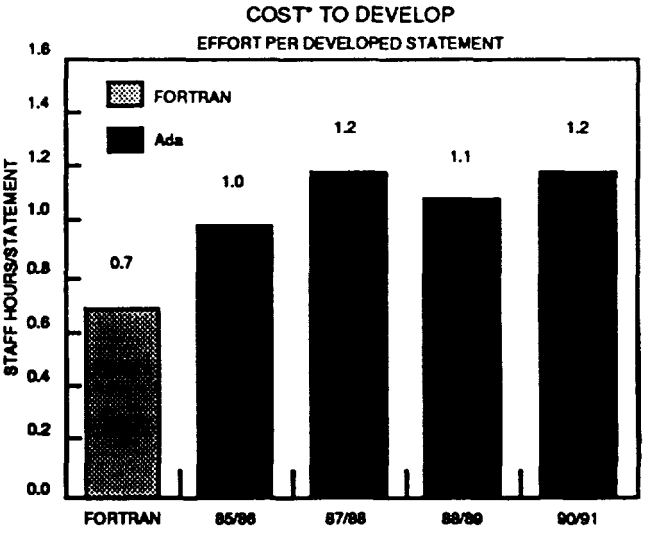
During this 6-year period, the SEL went through various levels of packaging the Ada/OOD methodology. On the earliest project in 1985, when OOD was still very young in the industry, the SEL found it necessary to tailor and package their own General Object-Oriented Development (GOOD) methodology [13] for use in the flight dynamics environment. This document (produced in 1986) adjusted and extended the industry standard for use in the local environment. In 1987, the SEL also developed an Ada Style Guide [14] that provided coding standards for the local environment. Commercial Ada training courses, supplemented with limited project-specific training, constituted the early training in these techniques. The SEL also produced lessons-learned reports on the Ada/OOD experiences, recommending refinements to the methodology.

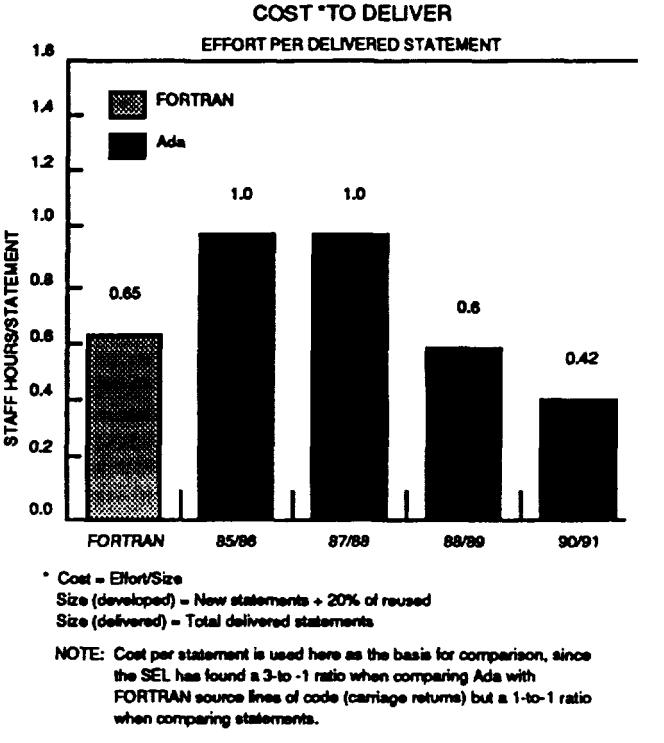
Recently, because of the stabilization and apparent benefit to the organization, Ada/OOD is being packaged as part of the baseline SEL methodology. The standard methodology handbooks [11, 12] include Ada and OOD as mainstream methods. In addition, a complete and highly tailored training program is being developed that teaches Ada and OOD as an integrated part of the flight dynamics environment.

Although Ada/OOD will continue to be refined within the SEL, it has progressed through all stages of the experience factory, moving from a candidate trial methodology to a fully integrated and packaged part of the standard methodology. The SEL considers it baselined and ready for further incremental improvement.

6. IMPLICATIONS FOR THE DEVELOPMENT ORGANIZATION

For 15 years, NASA has been funding the efforts to carry out experiments and studies within the SEL. There have been significant costs and a certain level of overhead associated with these efforts; a logical question to ask is "Has there been significant benefit?" The historical information strongly supports a very positive answer. Not only has the expenditure of resources been a wise investment for the NASA flight dynamics environment, but members of the SEL strongly believe that such efforts should be





* Cost = Effort/Size
Size (developed) = New statements + 20% of reused
Size (delivered) = Total delivered statements

NOTE: Cost per statement is used here as the basis for comparison, since the SEL has found a 3-to -1 ratio when comparing Ada with FORTRAN source lines of code (carriage returns) but a 1-to-1 ratio when comparing statements.

Figure 12. Costs To Develop and Deliver

commonplace throughout both NASA and the software communit in general. The benefits far outweigh the costs.

Since the SEL's inception in 1976, NASA has spent approximatel $14 million dollars (contract support) in the three major suppo areas required by this type of study environment: research (defir ing studies and analyzing results), technology transfer (producin standards and policies), and data processing (collecting forms an maintaining data bases). Approximately 50 staff-years of NAS personnel effort have been expended on the SEL. During this sam period, the flight dynamics area has spent approximately $150 mi lion on building operational software, all of which has been part the study process.

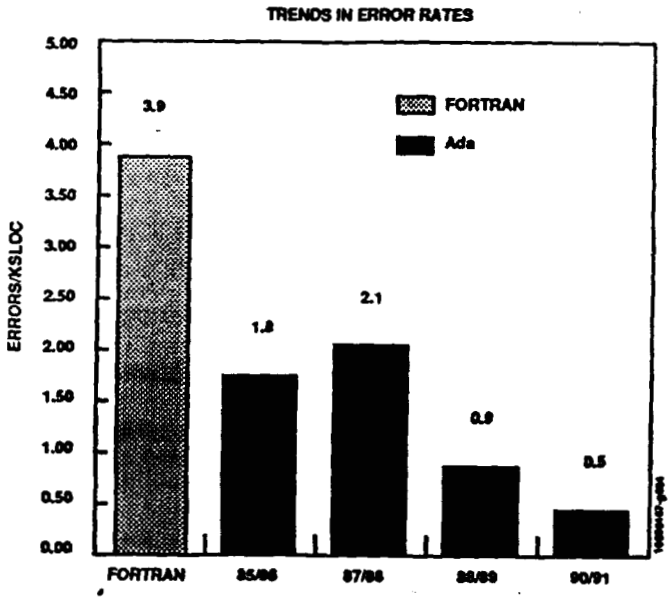


Figure 13. Trends in Error Rates

During the past 15 years, the SEL has had a significant impact on the software being developed in the local environment, and there is strong reason to believe that many of the SEL studies have had a favorable impact on a domain broader than this one environment. Examples of the changes that have been observed include the following:

1. The cost per line of new code has decreased only slightly, about 10 percent—which, at first glance might imply that the SEL has failed at improving productivity. Although the SEL finds that the cost to produce a new source statement is nearly as high as it was 15 years ago, there is appreciable improvement in the functionality of the software, as well as a tremendous increase in the complexity of the problems being addressed [15]. Also, there has been an appreciable increase in the reuse of software (code, design, methods, test data, etc.), which has driven the overall cost of the equivalent functionality down significantly. When the SEL merely measures the cost to produce one new source statement, the improvement is small; but when it measures overall cost and productivity, the improvement is significant.
2. Reliability of the software has improved by 35 percent. As measured by the number of errors per thousand lines of code (E/KSLOC), flight dynamics software has improved from an average of 8.4 E/KSLOC in the early 1980s to approximately 5.3 E/KSLOC today. These figures cover the software phases through acceptance testing and delivery to operations. Although operations and maintenance data are not nearly so extensive as the development data, the small amount of data available indicates significant improvement in that area as well.
3. The "manageability" of software has improved dramatically. In the late 1970s and early 1980s, the environment experienced wide variations in productivity, reliability, and quality from project to project. Today, however, the SEL has excellent models of the process; it has well-defined methods; and managers are better able to predict, control, and manage the cost and quality of the software being produced. This conclusion is substantiated by recent SEL data that show a continually improving set of models for planning, predicting, and estimating all development projects in the flight dynamics environment. There no longer is the extreme uncertainty in estimating such common parameters as cost, staffing, size, and reliability.
4. Other measures include the effort put forth in rework (e.g., changing and correcting) and in overall software reuse. These measures also indicate a significant improvement to the software within this one environment.

In addition to the common measures of software (e.g., cost and reliability), there are many other major benefits derived from a "measurement" program such as the SEL's. Not only has the understanding of software significantly improved within the research community, but this understanding is apparent throughout the entire development community within this environment. Not only have the researchers benefited, but the developers and managers who have been exposed to this effort are much better prepared to plan, control, assure, and, in general, develop much higher quality systems. One view of this program is that it is a major "training" exercise within a large production environment, and the 800 to 1000 developers and managers who have participated in development efforts studied by the SEL are much better trained and effective software engineers. This is due to the extensive training and general exposure all developers get from the research efforts continually in progress.

In conclusion, the SEL functions as an operational example of the experience factory concept. The conceptual model for the SEL presented in Section 1 maps to the functional groups discussed under SEL operations in Section 3. The experience base in Figure 2 is realized by the SEL data base and its archives of management models and relationships [16]. The analysis function from Figure 2 is performed by the SEL team of software engineering analysts, who analyze processes and products to understand the environment, then plan and execute experiments to assess and refine the new technologies under study. Finally, the synthesis function of the experience factory maps to the SEL's activities in packaging new processes and technology in a form tailored specifically to the flight dynamics environment. The products of this synthesis, or packaging, are the guidelines, standards, and tools the SEL produces to infuse its findings back into the project organization. These products are the experience packages of the experience factory model.

Current SEL efforts are focused on addressing two major questions. The first is "How long does it take for a new technology to move through all the stages of the experience factory?" That is, from understanding and baselining the current environment, through assessing the impacts of the technology and refining it, to packaging the process and infusing it into the project organization. Preliminary findings from the SEL's Ada and Cleanroom experiences indicate a cycle of roughly 6 to 9 years, but further data points are needed. The second question the SEL is pursuing is "How large an organization can adopt the experience factory model?" The SEL is interested in learning what the scaleup issues are when the scope of the experience factory is extended beyond a single environment. NASA is sponsoring an effort to explore the infusion of SEL-like implementations of the experience factory concept across the entire Agency.

ACKNOWLEDGMENT

Material for this paper represents work not only of the authors listed, but of many other SEL staff members. Special acknowledgment is given to Gerry Heller of CSC, who played a key role in editing this paper.

REFERENCES

Numerous papers, reports, and studies have been generated over the SEL's 15-year existence. A complete listing of these can be found in the *Annotated Bibliography of Software Engineering*

Laboratory Literature, SEL-82-1006, L. Morusiewicz and J. Valett, November 1991.

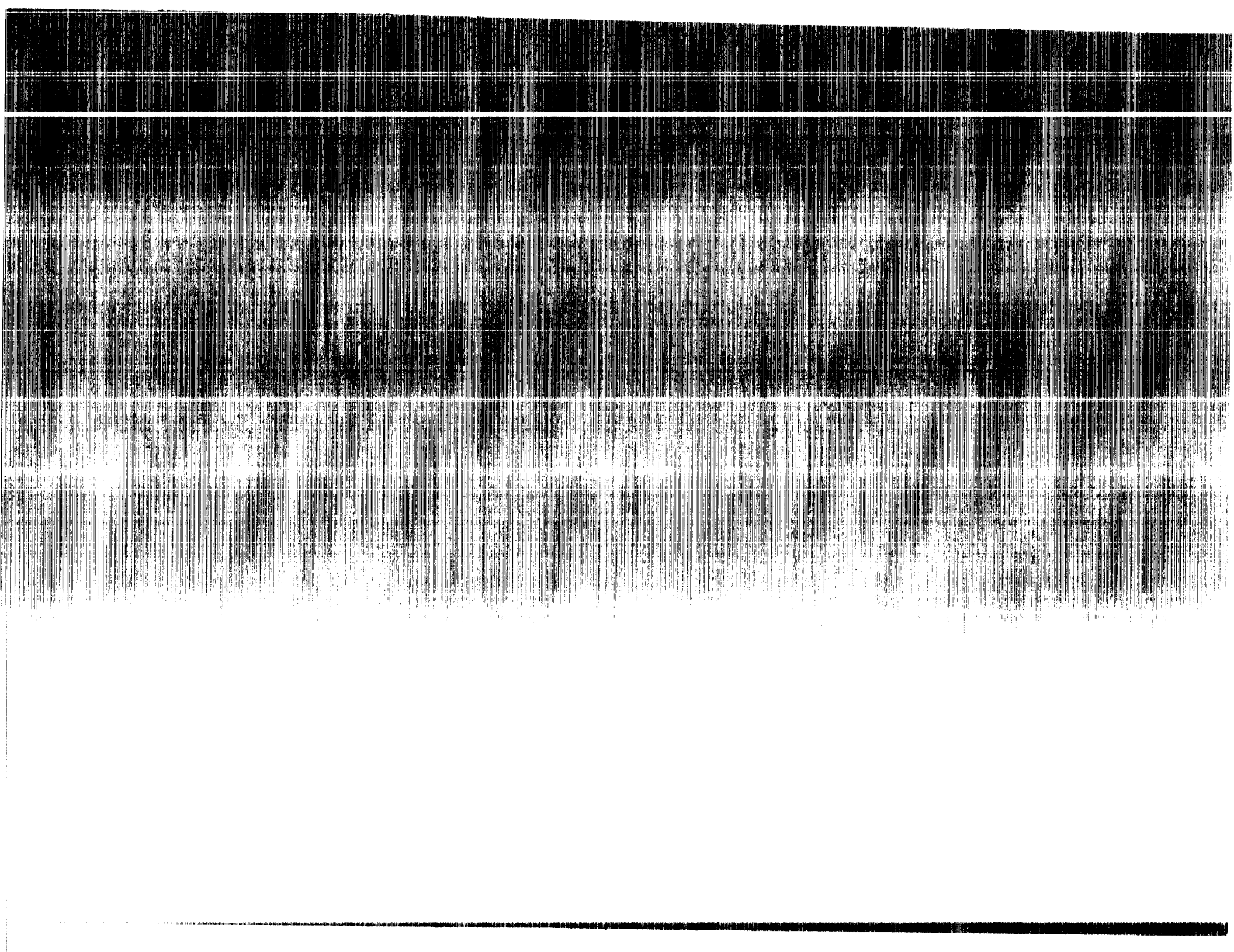
This bibliography may be obtained by contacting:

The SEL Library
Code 552
NASA/GSFC
Greenbelt, MD 20771

A listing of references specific to this paper follows.

1. V. R. Basili, "Towards a Mature Measurement Environment: Creating a Software Engineering Research Environment," Proceedings of the Fifteenth Annual Software Engineering Workshop, NASA/GSFC, Greenbelt, Maryland, SEL-90-006, November 1990.
2. V. R. Basili, "Quantitative Evaluation of a Software Engineering Methodology," Proceedings of the First Pan Pacific Computer Conference, Melbourne, Australia, September 1985.
3. V.R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984, pp. 728–738.
4. V. R. Basili, "Software Development: A Paradigm for the Future (Keynote Address)," Proceedings COMPSAC '89, Orlando, Florida, September 1989, pp. 471–485.
5. V.R. Basili and H.D. Romback, "Tailoring the Software Process to Project Goals and Environments," Proceedings of the Ninth International Conference on Software Engineering, Monterey, California, March 30 – April 2, 1987, pp. 345–357.
6. V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, Vol. 14, No. 6., June 1988, pp. 758–773.
7. V.R. Basili and G. Caldiera, "Methodological and Architectural Issues in the Experience Factory," Proceedings of the Sixteenth Annual Software Engineering Workshop, NASA/GSFC, Greenbelt, Maryland, Software Engineering Laboratory Series, December 1991.
8. V. R. Basili, G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 1, January 1992, pp. 53–80.
9. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," IEEE Software, November 1990, pp. 19–24.
10. S. Green, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, SEL-91-004, November 1991.
11. L. Landis, F. E. McGarry, S. Waligora, et al., *Manager's Handbook for Software Development (Revision 1)*, SEL-84-101, November 1990.
12. F.E. McGarry, G. Page, S. Eslinger, et al., *Recommended Approach to Software Development*, SEL-81-205, April 1983. Revision 3 in preparation; scheduled for publication June 1992.
13. E. Seidewitz and M. Stark, *General Object-Oriented Software Development*, SEL-86-002, August 1986.
14. E. Seidewitz et al., *Ada® Style Guide (Version 1.1)*, SEL-87-002, May 1987.
15. D. Boland et al., *A Study on Size and Reuse Trends in Attitude Ground Support Systems (AGSSs) Developed for the Flight Dynamics Division (FDD) (1976-1988)*, NASA/GSFC, CSC/TM-89/6031, February 1989.
16. W. Decker, R. Hendrick, and J. Valett, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, SEL-91-001, February 1991.

SECTION 3 – SOFTWARE TOOLS



SECTION 3—SOFTWARE TOOLS

The technical papers included in this section were originally prepared as indicated below.

- "Towards Automated Support for Extraction of Reusable Components," S. K. Abd-El-Hafiz, V. R. Basili, and G. Caldiera, *Proceedings of the IEEE Conference on Software Maintenance, 1991 (CSM 91)*, October 1991
- "Automated Support for Experience-Based Software Management," J. D. Valett, *Proceedings of the Second Irvine Software Symposium (ISS '92)*, March 1992

Towards Automated Support for Extraction of Reusable Components

S. K. Abd-El-Hafiz V. R. Basili G. Caldiera
Institute for Advanced Computer Studies,
Department of Computer Science,
University of Maryland, College Park, MD 20742, U.S.A.

Abstract

A cost effective introduction of software reuse techniques requires the reuse of existing software developed in many cases without aiming at reusability. This paper discusses the problems related to the analysis and reengineering of existing software in order to reuse it. We introduce a process model for component extraction and focus on the problem of analyzing and qualifying software components which are candidates for reuse. A prototype tool for supporting the extraction of reusable components is presented. One of the components of this tool aids in understanding programs and is based on the functional model of correctness. It can assist software engineers in the process of finding correct formal specifications for programs. A detailed description of this component and an example to demonstrate a possible operational scenario are given.

1 Introduction

Successful reuse of software resources can increase the overall quality and productivity in software projects by a large factor. Some of the problems that still limit software reuse are:

1. The difficulty of understanding a given software product in the absence of its original developers.
2. The scarce availability of reusable objects, even though there is a tremendous amount of available software.
3. The difficulty of retrieving, from a large data base, software components which can best match the given semantics requirements.
4. The lack of extraction and adaptation techniques that facilitate the reuse process.

New process models for software development should substitute the existing ones that are not defined to benefit from or support reuse. These new models should take advantage of reuse, introduce more reusable resources, and overcome the existing problems that limit reuse.

Developing reusable components is generally more expensive than developing specialized code, because of the overhead of designing for reusability and maintaining the component repository. A rich and well-organized catalog of reusable components is the key to a successful component repository and a long term economic gain. Moreover, such a catalog will not be available to an organization unless it can reuse the same code it developed in the past. Mature application domains, where most of the functions that need to be used already exist in some form in earlier systems, should provide enough components for code reuse. For example, Lanergan and Grasso found rates of reuse of about 60% in business applications[1]. A technique for extracting reusable components can improve productivity since it provides the software developer with components that are ready for reuse or need minor adaptation. Moreover, it can improve the software quality as it helps in better understanding these components during the extraction process.

In this paper, we use a process model[2] that serves not only to enhance the development of the project under consideration but also to organize and plan for better reuse technology in future projects. This model splits the traditional life-cycle model into two separate organizations, the project organization and the experience factory. In this framework we introduce a process model for component extraction and focus on the problem of qualifying candidate software components for reuse.

A prototype tool constituting one of the elements of an integrated system for extracting reusable compo-

nents is described. This prototype tool helps in understanding programs by deriving their specifications and is based on the functional model of correctness[3, 4]. The tool could be applied to program fragments as well as to complete programs and it helps in simultaneously checking syntax, static semantics, and generating specifications. We conclude the paper with an example to demonstrate a possible operational scenario of the tool.

2 Organizing the component extraction

Currently, all reuse occurs in the project development, where there is a completion deadline and the top priority is to deliver the system on time. This makes the objective of developing reusable software, at best, a secondary concern. Besides, project personnel cannot recognize the pieces of software appropriate for other projects.

We make use of a reuse-oriented model based on two separate organizations[2]:

- **The project organization:** Its goal is to deliver the systems required by the customer. The process model can be chosen based upon the characteristics of the application domain, taking advantage of prior software products and experience.

- **The experience factory:** It supports project development by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. Within the experience factory, we can identify various sub-organizations. One of them is the component factory which develops reusable components, extracts reusable components from existing systems, and generalizes or remodels any previously produced component.

Different conceptual architectures can be used for the component factory[5]. At one extreme there is the clustered architecture in which all software development activities are concentrated in the project organization and the component factory is dedicated only to processing already existing software. At the other extreme there is the detached architecture in which the development activities are concentrated in the component factory and the project organization performs only high-level design and integration. The clustered architecture is much closer to the way software is currently implemented. The development of the components is probably faster in the project organization since there is less communication overhead and more direct pressure for their delivery. On the other hand, the components developed are more context dependent. In the detached architecture, there is more emphasis on developing general purpose components in order to serve several project organizations more efficiently. On the other hand, there are more chances for bottlenecks and for periods of inactivity due to the lack of requests from the projects. The detached architecture is probably better suited for environments where the practice of reuse is formalized and mature. An organization that is just starting with reuse should probably instantiate its component factory using the clustered architecture and then, when it reaches a sufficient level of maturity and improvement with this architecture, start implementing the detached architecture in order to continue the improvement.

In any case, the extraction of reusable components is a characteristic activity of the component factory. The next section will present in detail the features of this activity, in the framework of a component factory. Caldiera and Basili[6] have proposed a process model for the extraction of reusable components in two phases: the identification phase and the qualification phase (see figure 1). The necessary human intervention in the second phase is the main reason for splitting the process in two steps. The first phase, which can be fully automated, reduces the amount of expensive human analysis needed in the second phase by limiting analysis only to components that really look worth considering.

3 The extraction process

3.1 Identification

Program units are automatically extracted and made to be independent compilation units. These independent units are measured according to observable properties related to their potential for reuse in three steps. These steps are summarized here:

1. Definition of the reusability attribute model: A set of automatable measures that captures the characteristics of potentially reusable components is defined along with acceptable ranges of values for these metrics.

2. Extraction of components: Modular units (e.g. C functions. Ada subprograms or blocks, or Fortran subroutines) are extracted from existing software and

completed so that they have all the external references needed to reuse them independently.

3. Application of the model: The current reusability attribute model is applied to the extracted, completed components. Components whose measures are within the model's range of acceptable values become candidate reusable components to be analyzed in the qualification phase.

A detailed description of the component identification phase, a definition of a basic reusability attribute model, and an application of this model on several case studies using a computer-based system have already been discussed in the literature[6].

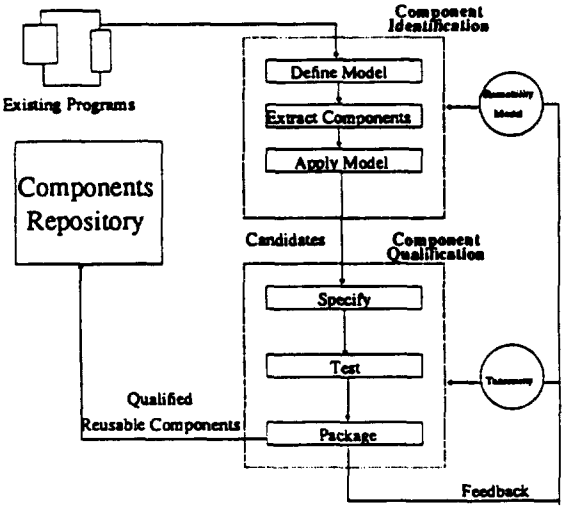


Figure 1: Component extraction.

3.2 Qualification

The extracted components are analyzed in order to understand them and record their meaning. The components are packaged by associating with them a reuse specification, a significant set of test cases, a set of attributes based on a reuse classification scheme, and a set of procedures for reusing the component. This phase consists of following steps:

1. Formal specification: A precise description of what the component does is generated and some assurance is obtained that the component meets the requirements.

Since formal specifications are based on mathematical notations, they help in understanding the software by removing the ambiguities which might be introduced by any informal notation. Formal specifications are different from the programs they specify since they only express the behavior of the program without stating how the program derives this behavior. So, formal specifications are the basis for selecting and storing software components as they improve understandability and assist in producing more reliable and higher quality software. Since the specification of complex tasks may in itself be complex, the process of specification construction must be formalized and supported by automated tools. In the next section, we will describe a prototype tool that aids in understanding programs. This tool provides automated support for deriving the functional specifications of programs and proving their partial correctness. In other words, it helps in proving that the program is consistent with its specification but does not prove its termination.

Formally specifying a software component and proving its partial correctness do not mean that the component will pass this step. There are several other properties that should exist in the candidate components for the sake of understandability. We must not ignore other important features such as proper documentation, use of meaningful variable names, and the structured style of programming. The informal information that the software engineer deals with cannot be ignored relying on the fact that the automated specifications tools will supplement those features. The informal information is important in explaining some intuitive ideas that are hard to explain using formal specifications.

Since we need both formal and informal information, a domain expert is needed to perform the specification step. This expert extracts the formal specification of each candidate reusable component, assisted by the automated tools available, and examines the other informal features that cannot be judged using automated tools. Components that are not relevant, not correct, or whose functional specification is not easy to extract are discarded. The expert reports reasons for discarding candidates and other insights that will be used to improve the reusability attributes model.

2. Testing: Test cases are generated, executed and associated with components. Deriving the functional specification and proving the correctness of a program do not mean that it will not fail when compiled and/or executed. This might simply be due to the fact that termination of the program has not been proven. Moreover, in most verification and specification systems, arithmetic operations ignore things such as overflow, underflow, and round-off errors.

Testing can take advantage of the functional specification generated by performing functional testing. Also, structural testing can be done using a coverage analyzer. If, as is likely, the component needs a

'wrapping' to be executed, the testing step generates this wrapping. If a component passes the testing then test cases, wrapping, and test results are stored in the component repository. Components that do not satisfy the test are discarded. Again, the reasons for discarding candidates are recorded and used to improve the reusability attributes model and possibly the process for extracting the functional specification. This is most likely the last step at which a component will be discarded.

3. **Packaging:** The extracted candidates are stored in the component repository along with their functional specifications and test cases. The component repository is actually a data base of experience in which information on software products, processes, and measures of aspects of them is stored. That is why we organize this data base by classifying both the reusable components and their development histories according to several domain dependent criteria.

Information for the future reuser is provided in a manual that contains a description of the component's function and interfaces as identified during generation of its functional specification, directions on how to install and use it, information about its procurement and support, and information for component maintenance.

At the end of each process cycle the reusability attribute model is updated by drawing on information from the qualification phase to add more measures, modify or remove measures that proved ineffective, or alter the range of acceptable values. This step requires analysis and possibly even further experimentation. The taxonomy is updated by adding new attributes or modifying the existing ones according to problems reported by the experts who classify the components.

4 The CARE system

4.1 Overview

The CARE[6] system(CARE[1]: Computer Aided Reuse Engineering) has been designed to support the proposed process model for extracting reusable components. As shown in figure 2, it consists of two main subparts: the component identifier and the component qualifier. The component identifier consists of the model editor, which helps in defining and modifying the reusability attributes model, and the component extractor which applies such model to the programs. The component qualifier consists of the specifier, the tester, and the packager. The current version of the CARE system consists of the component extractor and the specifier. It runs on a Sun Workstation and supports ANSI C and Ada. In the rest of this section we focus on the description of the specifier.

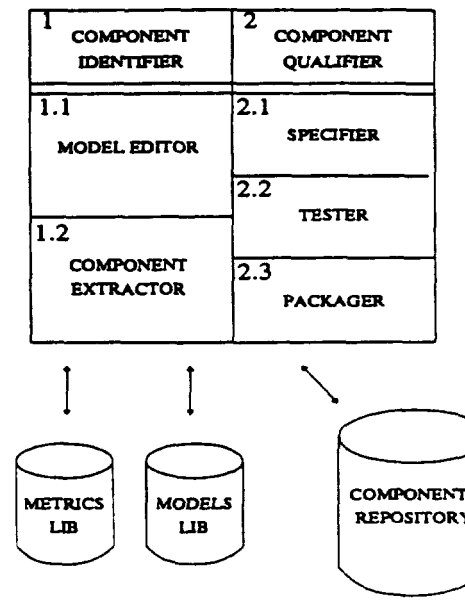


Figure 2: CARE system architecture.

4.2 The component specification tool

The prototype specifier included in the CARE tool is the second in a series of prototype tools developed at the Computer Science Department of the University of Maryland under the general name FSQ, for Functional Specification Qualifier. This prototype supports the derivation of programs specifications and the verification of whether or not the programs meet those specifications. It does not only help to specify and check the partial correctness of finished programs, but it also works on unfinished programs and program fragments. It is a program understanding tool that is based on a formal specification technique. CARE-FSQ$_2$ uses Mills' functional model of correctness[3, 4] in order to derive the specifications. This model requires the user to provide only the loop function and then a technique is provided to derive the program specification. Other techniques[7, 8] require the user to provide an entry assertion, an exit assertion, and a loop assertion. Those techniques are more useful in verifying that the program is consistent with its specification. The process of deriving specifications helps more in understanding the software. Moreover, the functional method pro-

[1]The CARE system is under development at the Computer Science Department of the University of Maryland

vides simple and intuitive notations that can be easily understood.

The CARE-FSQ$_2$ prototype helps in checking syntax, static semantics, and generating specifications at the same time. CARE-FSQ$_2$ also provides the capability of carrying out some algebraic simplifications and enables the user to make use of some well defined mathematical functions in the specification of the loop function.

4.2.1 Formal foundation: Each statement S is given a meaning as a function from a program state to another state. A state is a mapping from the variable names to their current values. The square bracket notation is used to denote the function represented by the program construct contained inside the brackets, i.e. $[S]$ represents the function computed by the statement S. We use four basic structures[3, 4]:

1. Assignment

The meaning of the assignment $v := e$, where v is a variable and e is an expression, is:

$$[v := e] = \{(S,T) : T = S \text{ except that } [v](T) = [e](T)\}$$

We can define the meaning of variables and expressions as a mapping from a state to a value.

2. Composition

If A and B are statements and $\circ$ is functional composition, we have:

$$[A;B] = [A] \circ [B]$$

3. Alternation

$$[\ if\ B\ then\ S\ fi\ ] = \{(U,[S]U) : [B](U) = true\} \cup \{(U,U) : [B](U) = false\}$$

$$[if\ B\ then\ S_1\ else\ S_2\ fi] = \{(U,[S_1]U) : [B](U) = true\} \cup (U,[S_2]U) : [B](U) = false\}$$

4. Iteration

$$[while\ B\ do\ S\ od] = \{(T,U) : \exists k \geq 0 : \forall 0 \leq i < k\ (([B]([S]^i(T)) = true \wedge [B]([S]^k(T)) = false \wedge [S]^k(T) = U)\}$$

In other words, the loop function is undefined for a state T unless there is a natural number k which denotes the number of iterations after which the test first fails. T is then transformed to the k-fold composition of S on T. In order to carry out practical proofs, the following characterizing theorem is needed[9].

Theorem

Let W be the program fragment *while B do S od*, Then $f = [W]$ if and only if:

1. $domain(f) = domain([W])$
2. $([B](T) = false) \implies f(T) = T$
3. $f = [if\ B\ then\ S\ fi] \circ f$

This theorem provides a method for deriving the correct loop function f:

1. Guess or work out a trial function f.
2. Use the three conditions of the theorem to check that the trial function is correct.

A trace table can be used to organize the derivation of program meanings (by a symbolic execution of the program)[4, 9].

The strength and weakness of the functional method, in comparison with other specification techniques, originate from the fact that even though exact functions state accurately the meaning of a loop, they are harder to work with than the weak assertions that suffice when there is a loop initialization providing a precondition.

4.2.2 The implementation: CARE-FSQ$_2$ is implemented using the Synthesizer Generator[10] and Maple, an interactive algebraic symbolic executor[11]. An overview of the tool is shown in figure 3. The Synthesizer Generator requires as an input a description of an attribute grammar and generates from it a hybrid language-based editor that allows a combination of text editing and structure editing. As the user edits program text and annotations, the system creates and edits abstract syntax trees that represent pieces of programs and their specifications. The attributes of the nodes of this tree carry information about the static semantics of the program as well as its specifications, and they are evaluated incrementally. The basic feature of Maple is its ability to simplify expressions involving unevaluated elements. As each complete statement is entered by the user, it is evaluated and the results are printed on the output device. Maple enables carrying out algebraic simplifications during the symbolic execution. In order to overcome the limitations of Maple in the evaluation of boolean expressions, CARE-FSQ$_2$ has an interactive feature that allows the user, before writing the specifications, to simplify boolean expressions and the expressions containing array notations.

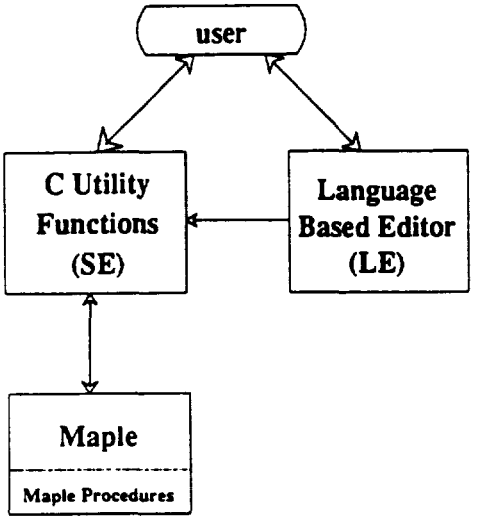


Figure 3: Overview of CARE-FSQ$_2$.

In a typical CARE-FSQ$_2$ session, the user derives the specifications of the program using step-wise abstractions. In other words, the user starts by trying to find the correct specification of every loop in the program as a separate entity. After succeeding in this, the correct specification of the whole program can be found. This methodology of step-wise abstraction enables the software engineer to concentrate on small pieces of code, one at a time, and to mitigate in this way the difficulty of specifying the whole program.

Currently, CARE-FSQ$_2$ supports a subset of Ada with modifications on the input/output mechanism. The data types supported are integer, boolean, character, a restricted form of floating point, constrained arrays, and user defined data types. The basic control structures of Ada are supported except unconditional 'go to' statements, and case statements. Static semantic checking is also included. A brief description of the input/output mechanism and the specification language is given in the rest of this subsection.

Input and output is done through atomic and stream ports[12]. A subprogram, called an elementary process, accepts input data from input ports, performs computation specified with an Ada-like notation, and returns results through output ports. The input and output of single data items can be carried out through atomic ports. Stream ports are used as schemes for data types whose elements can be accessed in a linear order. The stream ports of one process can be bound to particular data types to produce the implementation. Input and output ports can be bound to files to communicate with the system. This form of data abstraction helps in making the specification process more general and easier. The following seven operations are defined for atomic and stream ports:

1. Receive(p): To Receive a value via the input port p from the source associated with the port.
2. Send(p): To Send a value via the output port p to the destination associated with the port.
3. Initialize(p): To open the stream associated with the stream port p for reading.
4. Receive(p, v): To receive a value into a variable v from the stream associated with the input port p.
5. Send(p, v): To send the value of variable v to the stream associated with the output port p.
6. isEOS(p): A boolean function to check if end of stream is reached in the input stream port p.
7. Finalize(p): To close the stream associated with the port p. The effect of finalization for an output stream port is that the function isEOS becomes true at the consumer process.

The specifications for CARE-FSQ$_2$ are written using guarded command sets whose syntax is:

< *guarded command set* > ::=
    < *guarded command* >
    { | < *guarded command* >}

< *guarded command* > ::=
    < *boolean expr* > →
    < *concurrent assignment* >

< *concurrent assignment* > ::=
    < *var* > := < *expr* > | < *var* > ,
    < *concurrent assignment* > , < *expr* >

A concurrent assignment is an extension of the assignment statement where a number of different variables can be substituted simultaneously. The concurrent assignment statement is denoted by a list of different variables to be substituted at the left hand side of the assignment operator and an equally long list of expressions as its right hand side. The ith variable from the left hand list is to be replaced by the ith expression from the right hand list. The expressions can include calls to some mathematical functions such as min, max, product, sum, factorial, igcd (greatest common divisor), irem (remainder), and iquo (quotient).

An array is considered to be a partial function from subscript values to the type of array elements. The command $a(i) := e$ assigns a new function to a, a function that is the same as the old one except that at the argument i its value is e. The notation (a, i, e) is used to denote the array that is the same as a except when applied to the value i yields e. The notation $(a, index = m..n, e)$ is used to denote the array that is the same as a except when applied to index values between m and n, i.e. $m \leq index \leq n$, it yields e. The expression e can be a function of the bound variable $index$. To make the two notations consistent, (a, i, e) is written $(a, index = i, e)$ where $index$ is a bound variable. The notation defined for arrays are used for stream ports as well. A stream port is treated as an array whose subscript is of type integer with the first element subscript being one.

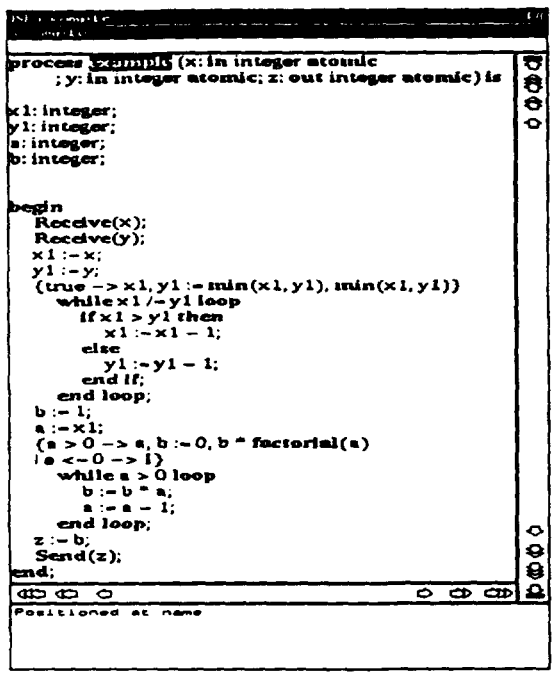


Figure 4: The program to be specified.

4.2.3 **Example:** We describe a short example, due to the space limitation, to demonstrate a sample result obtained using CARE-FSQ$_2$. In order to find the correct specification of a while loop, the user should annotate it with a trial loop function enclosed between two curly braces. CARE-FSQ$_2$ assists the user in verifying the correctness of the loop specification by calculating the composition $[if\ B\ then\ S\ fi] \circ f$. The user, on the other hand, must ensure that the three while loop verification conditions are satisfied. After verifying all the while loops in the program, the user can proceed to find the functional meaning of the whole program.

```
expr : (x1-y1 < 0 or y1-x1 < 0) and y1-x1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 < x1

expr : (x1-y1 < 0 or y1-x1 < 0) and not y1-x1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 > x1

expr : not (x1-y1 < 0 or y1-x1 < 0)
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 = x1


The symbolic execution result is :
==================================

 y1 < x1 ->
    x1, y1 :=
    min(x1-1,y1), min(x1-1,y1)

|

 y1 > x1 ->
    x1, y1 :=
    min(x1,y1-1), min(x1,y1-1)

|

 y1 = x1 ->
    x1, y1 :=
    min(x1,y1), min(x1,y1)
```

Figure 5: Finding the specification of the first loop.

```
expr : -a < 0 and -a+1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a > 1

expr : -a < 0 and a-1 <= 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a = 1

expr : not -a < 0 and a <= 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a <= 0


The symbolic execution result is :
==================================

 a > 1 ->
    a, b :=
    0, b*GAMMA(a+1)

|

 a = 1 ->
    a, b :=
    a-1, b*a

|

 a <= 0 ->
    a, b :=
    a, b
```

Figure 6: Finding the specification of the second loop.

Figure 4 shows a program that receives two integers as input , finds their minimum, calculates its factorial

if it is positive, and then saves the result in z. First, the verification conditions of the two while loop have to be checked. Hence, we let CARE-FSQ$_2$ print the composition $[if\ B\ then\ S\ fi] \circ f$ to assist us in this process. Before printing the results of the composition, the user is prompted to enter his simplifications for some expressions if he/she desires(see figures 5 and 6).

Since the three verification conditions are satisfied for both loops, we can therefore proceed to find the functional meaning of the whole program which is shown in figure 7.

```
The symbolic execution result is :
==================================

 -min(x,y) < 0 ->
    x, y, z, x1, y1, a, b :=
    x, y, GAMMA(min(x,y)+1), min(x,y),
    min(x,y), 0, GAMMA(min(x,y)+1)

|

 min(x,y) <= 0 ->
    x, y, z, x1, y1, a, b :=
    x, y, 1, min(x,y), min(x,y), min(x,y), 1
```

Figure 7: Specification of the whole program.

5 Conclusion

In this paper, we have presented a process model for extracting reusable components. It first identifies these components using software metrics, then it qualifies them. We have focused on the qualification phase which generates their formal specifications, generates a significant set of test cases, and packages them for future reuse. We have then described the specification tool of the qualification phase, CARE-FSQ$_2$, that helps in understanding programs by generating their correct formal specifications. Further research needs to be done in order to be able to qualify and tailor large programs for reuse.

Acknowledgement

Research for this study was supported in part by NASA (Grant NSG-5123), ONR (Grant NOOO14-87-k-0307), and Italsiel S.p.A. (IAP Grant).

References

[1] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Design and Code", *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, Sept. 1984, pp. 498-501.

[2] V. R. Basili, "Software Development: A Paradigm for the Future", *Proc. Compsac'89*, IEEE Computer Soc. Press, Los Alamitos, Calif., Order No. 1964, pp. 471-485.

[3] H. D. Mills, "The New Math of Computer Programming", *Communications of ACM*, vol. 18, no. 1, Jan. 1975, pp. 43-48.

[4] J. D. Gannon, R. B. Hamlet and H. D. Mills, "Theory of Modules", *IEEE Trans. on Software Engineering*, vol. SE-13, no. 7, July 1987, pp. 820-829.

[5] V. R. Basili, G. Caldiera, G. Cantone, "A Reference Architecture for the Component Factory", *Technical Report CS-TR-2607*, Institute for Advanced Computer Studies and Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742, March 1991.

[6] G. Caldiera and V. R. Basili, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Feb. 1991, pp. 61-70.

[7] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of ACM*, vol. 12, no. 10, Oct. 1969, pp. 576-580, 583.

[8] E. W. Dijkstra, "A Discipline of Programming", Prentice Hall, 1976.

[9] H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet, "Principles of Computer Programming: A Mathematical Approach", Boston, MA, Allyn and Bacon, 1987.

[10] T. W. Reps and T. Teitelbaum, The Synthesizer Generator Reference Manual, Springer-Verlag, 1989.

[11] B. W. Char et al, Maple User's Guide, Watcom Publication Limited, Waterloo, Ontario, 1985.

[12] B. Joo, "Adaptation and Composition of Program Components", Ph.D. Dissertation, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, 1990.

53-61

Automated Support for Experience-Based Software Management

P-19

N93-1[illegible]164

Jon D. Valett

NASA/Goddard Space Flight Center
Software Engineering Branch Code 552
Greenbelt, MD 20771
internet jvalett@gsfcmail.nasa.gov
phone: (301) 286-6564
FAX: (301) 286-9183

Abstract

To effectively manage a software development project, the software manager must have access to key information concerning a project's status. This information includes not only data relating to the project of interest, but also, the experience of past development efforts within the environment. This paper describes the concepts and functionality of a software management tool designed to provide this information. This tool, called the Software Management Environment (SME), enables the software manager to compare an ongoing development effort with previous efforts and with models of the "typical" project within the environment, to predict future project status, to analyze a project's strengths and weaknesses, and to assess the project's quality. In order to provide these functions the tool utilizes a vast corporate memory that includes a data base of software metrics, a set of models and relationships that describe the software development environment, and a set of rules that capture other knowledge and experience of software managers within the environment. Integrating these major concepts into one software management tool, the SME is a model of the type of management tool needed for all software development organizations.

Keywords: software management, measurement, reuse of experience, management tools

1.0 Background

Good software management is generally viewed as a critical ingredient in successful software projects. One key aspect of good management is having access to the data that are necessary to understand the strengths and weaknesses of an ongoing development effort. To provide such access, a myriad of management-oriented tools have been developed. These tools typically allow the software manager to perform cost and size estimation, to plan a development project, to set up work-breakdown structures, and to provide other planning needs. Such tools are certainly useful, yet they do not provide the full scope of functionality required for a manager to effectively evaluate a software project.

Ideally, an experience-based software management tool would enable a manager to observe

a project's progress, to compare that progress with other projects or with a model of how a project "normally" behaves, to predict key project parameters such as size, completion date, or errors, to assess the project's progress pointing out its strengths and weaknesses, and to analyze the quality of the software project and the software product. In order to provide this functionality, the tool would require access to key data relating to a project's status and to the past experience necessary to understand and manage the ongoing project. Included in this knowledge and experience is a data base of software metrics, a set of models of a development environment, a set of management rules that provide insight into a project's strengths and weaknesses, a set of quality definitions, and a set of relationships that help to define an environment's characteristics. Such a management tool would integrate this experience into a single environment providing the functionality required to actively monitor a software project.

A working model of the management tool described above is being developed within the Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center (GSFC). This tool, called the Software Management Environment (SME) uses software measurement and the experience acquired from software measurement as its basis. Other tools either are being or have been developed that utilize measurement as a major component. These tools include TAME [1], Amadeus[2], and GINGER[3]. SME is a unique experience-based tool because it focuses on utilizing the measurement and the experience of a measurement program to automate support for project managers in actually monitoring the progress of their projects. While the SME has been constructed for a specific development environment, the concepts, architecture, and functionality of the tool, which are described in this paper, are general enough for any organization to build a similar tool. This paper will discuss the management activities that the SME addresses, the components needed to build an SME, and how these components are integrated to provide the management functions described.

2.0 Management Activities

In order for the SME to be an effective tool, it must automate key management functions. While the current SME is not comprehensive in its coverage of all management functions, it does provide support for many important aspects of software management. The SME utilizes a measurement-based approach to software management. Within this approach reusing management experience is viewed as an important aspect of the management process. This experience-based approach to management includes the following activities:

Observation and Comparison: The manager monitors the progress of a project by examining key project measures such as effort, size, and errors. The manager compares the status of the current project with past projects and with models of these measures that represent the nominal case within the environment. By observing and comparing, the manager is able to determine the current project's status and the differences between the current project and the normal project within the environment.

Prediction and Estimation: The manager estimates key project parameters such as project cost and size. The manager also, uses various models and relationships to continually update these predictions. These activities allow the manager to determine at-completion values for important measures and to estimate project schedule.

Analysis: Based on the measurement data, past project experience, and subjective information about a project, the manager identifies potential project problems.

Assessment: Using available measurement data and definitions of project quality, the manager assesses the overall quality of the ongoing project. For example, these quality assessments provide the manager with an idea of the project's maintainability, correctability, and stability.

A software tool should only attempt to automate aspects of a process that are understood well enough to perform manually; in the case of SME, all of the activities described above are carried out on projects within this development environment. In fact, such activities are part of the normal management process. The SME integrates data and experience into one tool that provides managers with functions that help them to perform these activities.

3.0 The Software Management Environment (SME)

The Software Engineering Laboratory (SEL) has actively been developing the management concepts that are the basis for the SME for the past 15 years. A prototype of the tool was developed between 1984 and 1987; this prototype provided a set of recommendations for developing an actual version of the tool.[4] This set of recommendations was then incorporated into the actual development of the SME, which began in 1987. The remainder of this section will discuss the SEL and the concepts that are the underlying ideas for the SME.

3.1 The Software Engineering Laboratory

The SEL was established in 1976 and has three primary organizational members: NASA/GSFC, Software Engineering Branch; The University of Maryland, Computer Science Department; and The Computer Sciences Corporation, Software Engineering Operation. The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices.[5] During the SEL's 15 years it has collected data on over 100 software development projects. These data include such items as software development effort, software size, error data, change data, and computer utilization data and are stored in a large repository called the SEL data base.[6] This data base has been

used throughout the past 15 years to help the SEL to accomplish its three objectives. In the process of studying and measuring this particular development environment the SEL has produced numerous reports and papers which characterize this environment, evaluate various tools and methods, and capture experience and lessons learned in various software development efforts. (For a complete list of SEL documents and reports see the "Annotated Bibliography of Software Engineering Literature".[7])

Throughout the SEL's history, this software measurement program has been used extensively in the management of actual software projects. Such use of measurement data is common among companies that have instituted measurement programs (eg. reference [8]). As this use of measurement as a management tool evolved, the SEL began attempts to automate the process. Such automation is only possible through a comprehensive understanding of how to use software measurement data within a particular development environment. Within the SEL environment, software managers use not only the data collected on their current project, but also, the information and experience from past projects. The studies and reports characterizing the environment provide the manager with profiles of how particular measures behave, numerous relationships for estimation and prediction of such measures, and lessons learned concerning how to analyze measurement data. Automating the access to this vast corporate resource is the goal of the SME.

3.2 SME Concepts

Understanding the SME requires a firm understanding of the three major components that are the basis for the tool. The first is the SEL data base, it provides the historical data of past projects, as well as the dynamic data on projects that are currently being managed. The second, is a set of models and relationships that describe the development environment. These models and relationships provide the profile of a normal project, as well as the necessary information to predict and estimate key project parameters. Finally, experienced software managers analyze

measurement data to determine a project's strengths and weaknesses. The knowledge required to perform this analysis is captured in management rules that provide the expert analysis portion of the SME. These three SME concepts provide the experience base needed for an organization to construct an SME-like tool.

An important aspect of these SME concepts is that the experience they represent continually evolves as the development environment and process changes. The SME packages the current level of experience; as it changes, the experience base is refined to reflect these changes. The representation of the experience, however, does not change. Therefore, the key aspect of the SME, from the perspective of someone who wishes to build a similar tool, is the concepts and the architecture of those concepts, not the experience itself.

Software Measurement Data

Measurement of the software development process and its products is a necessary component of successful software management. Within the SME, data from the SEL data base is utilized to provide the underlying measurement data. The SEL data base captures information on all software projects within one particular development environment. This data includes such items as the weekly effort expended on a project, the size of the ongoing software project (in both lines of code and number of modules), the amount of computer utilization on a project, and the number of errors uncovered as well as the number of changes made to the source code. In addition to these basic measures, the SEL data base contains data on such items as number of modules designed, number of open problem reports, and the amount of time spent uncovering and repairing errors. While these lists of data are not complete, they do provide a snapshot of the types of data available to the SME.

The SME uses the data from the SEL data base as a basis for all of its analysis, comparison, prediction and assessment. The data provide the information that characterize and describe the current software development project as well as past projects of interest. Having access to so much descriptive data allows the SME to provide its wide range of functionality. Thus, software

measurement is the backbone of the SME. Measurement provides the basis for all other SME concepts; neither the management rules nor the models and relationships would be possible without it.

Models and Relationships

The second component of the SME is the models and relationships that represent the software development process and its products. The models and relationships used within the SME and presented within this paper are derived from numerous previous SEL reports and studies. A summary of the types of models and relationships used can be found in the document "The Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules".[9]

The term model is used to describe a pattern of how some measure or combination of measures normally behaves within a software development environment. Measurement models have been described in numerous SEL reports and papers, but they have generally all been developed using similar methods. Typically, a model for some particular measure is developed by examining the data for that measure over a set of similar projects. The data is then combined, usually using some type of averaging, to develop a model of the "normal" project. Since even within one environment all projects may not be homogeneous, different models for the same measure are developed for significantly different project types. Within the SME, there are currently two different model types, depending on the development methodology used on the projects. Other models may need to be developed depending on such parameters as project type, programming language, or development environment. Deciding what different factors constitute a distinct model type is an important research component of developing an SME. Certainly, each individual project is distinct, but usually projects within a development environment have many similarities that result in reasonable models.

As an example of a model that is used by SME, Figure 1 shows how source code grows within the SEL environment. (For the purposes of this paper, there is no need to distinguish between various model types.) It provides a representation of the typical growth of the number

of source lines of code within a project's controlled library. The wide band indicates a range of what is considered to be "normal" source code growth. (In this case the range is one standard deviation on either side of the actual model.) As another example, figure 2 is the model of error rate for the SEL environment. This model shows the typical errors uncovered and repaired per line of code within the environment throughout a project's lifetime. Again, the band represents a range over which the error rate is considered "normal." (In both Figures 1 and 2, lines of code is defined as physical lines including commentary and blank lines. In Figure 2, error is defined as a conceptual error in the software.) Another kind of model used within the SME is of the amount of time spent in each phase of a project. This model is depicted in Figure 3; it provides a mechanism for determining how much calendar time a project normally spends in each phase of the software development life cycle.

Relationships, on the other hand, provide the SME with a way to estimate critical project factors based on other estimates, or current status. Relationships are typically developed by using numerous software development projects' data to determine if any correlation exists between various measures. Normally, such data analysis is done to test hypotheses that certain relationships exist between such measures.

As an example, within the SEL environment, a relationship has been found between lines of code and the actual duration of a project. This relationship is shown as the equation:

$$D = 5.450 * L ** 0.203$$

where,

D is the duration of the project in months (from project start through acceptance test), and

L is the total delivered lines of code in thousands.

Such a relationship allows a manager to estimate the length of a project based on an estimate of the number of lines of code for that project. Other relationships have been established between computer use and lines of code, effort and number of modules, etc. Such relationships provide a software manager both a mechanism for estimating various parameters and a consistency check for sets of estimates.

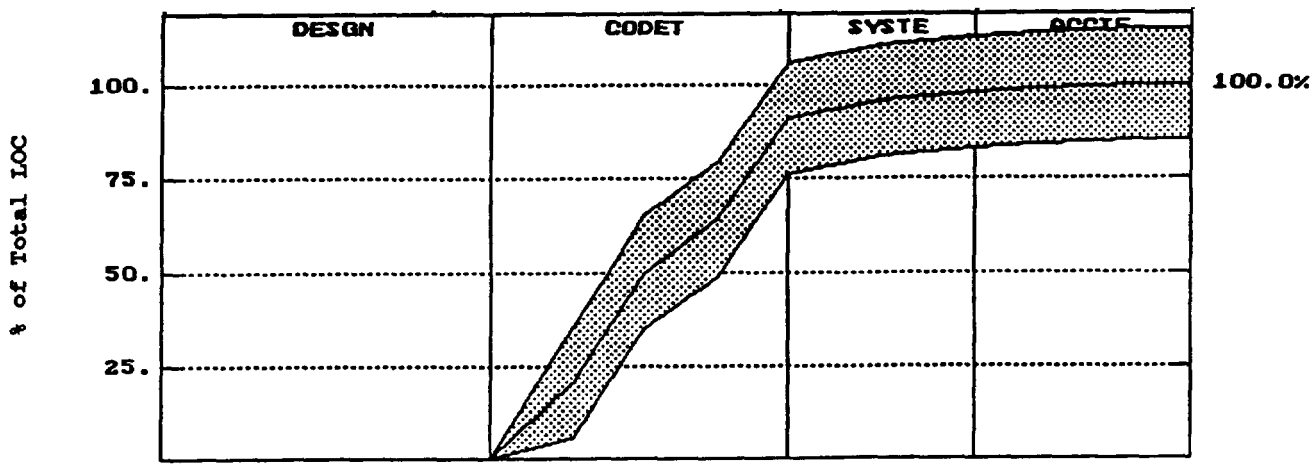


Figure 1: Model of Source Code Growth

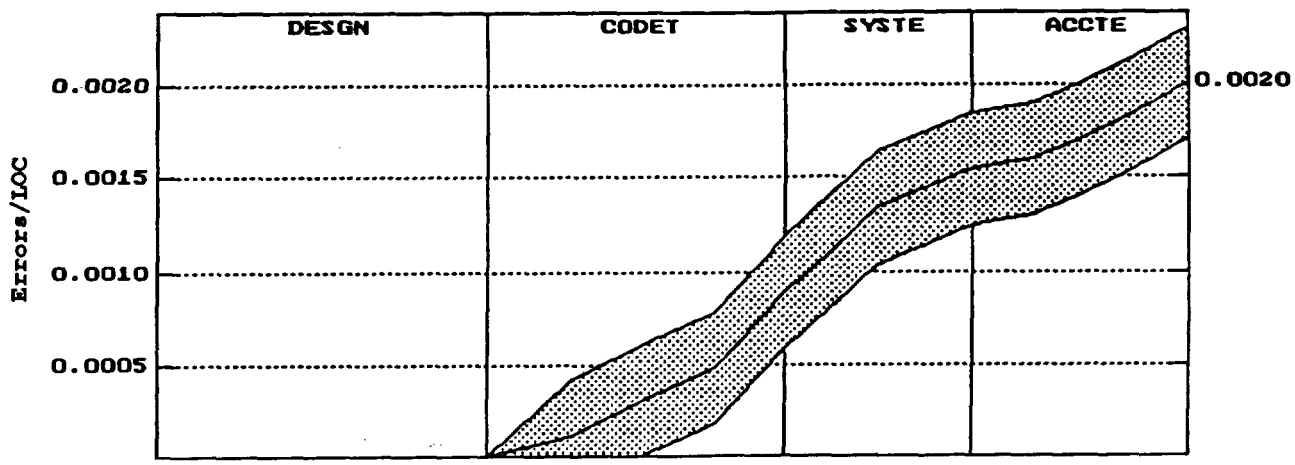


Figure 2: Model of Errors/Line of Code

Management Rules

Capturing how experienced software managers use and evaluate measurement data has been investigated by the SEL.[10] These studies show that using expert systems techniques for the capture and use of this experience is feasible in this domain. This knowledge about software measurement has been published in numerous SEL reports and it provides a foundation for creating an experience base for utilizing software measures in management.[9] The concept of these software management rules is that interviewing software managers and capturing how they interpret certain conditions of a project provides reusable knowledge concerning the strengths and weaknesses of a project. These interpretations are then combined into specific management rules that describe the possible explanations for certain conditions. For example, figure 4 shows a graphic of a simple management rule. This figure shows how one might interpret a deviation from the normal pattern of computer use per line of source code (again represented as a model similar to those described in the previous section). For example, early in the project if the number of CPU hours per line of code is above normal one possible interpretation is that the design was not actually complete. Later in a project, if the measure is below normal, the possible explanations might be either low productivity, or insufficient testing. Such a figure provides a simple representation of a management rule.

Actually, a number of simple management rules can be combined to form rules that describe the possibilities that certain explanations are true. For example, a rule such as

If the number of programmer hours per software change is above normal and
the project is early in the code phase then possible explanations are
Good solid, reliable code (0.5)
Poor testing (0.25)
Changes are hard to isolate (0.25)
Changes are difficult to make (0.25).

describes the possible explanations for a certain condition. This rule uses numbers to show the

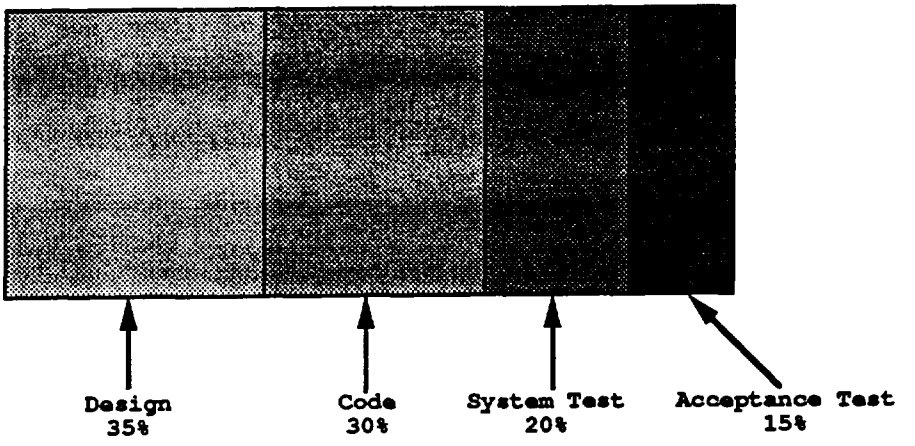


Figure 3: Model of Project Schedule

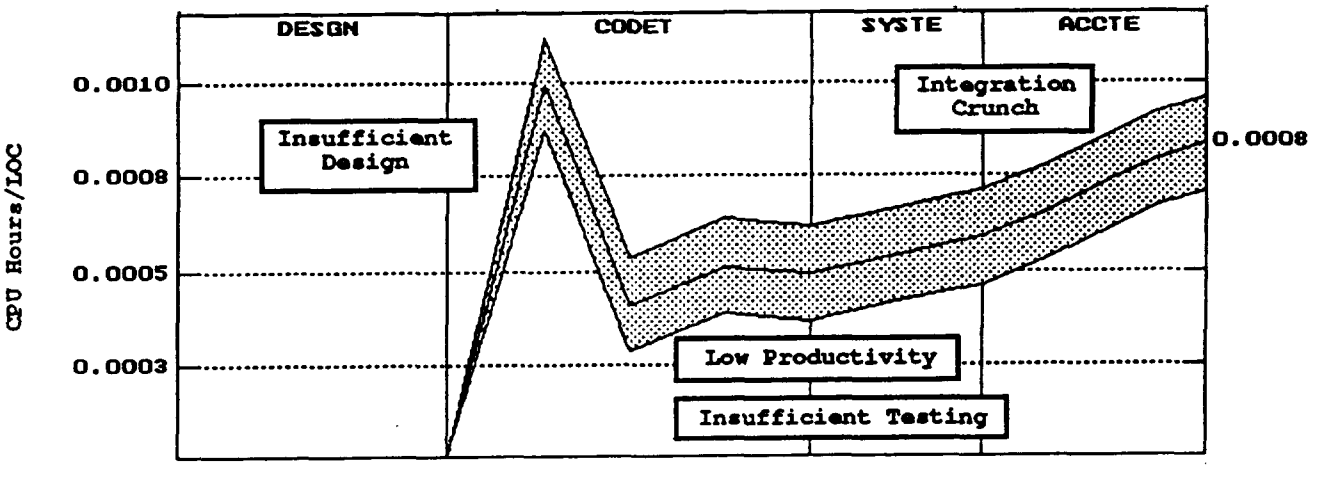


Figure 4: Rule for Analyzing Computer Use

certainty that each of the possible consequents are true. Thus, it is more likely that good solid, reliable code is the explanation for the deviation then poor testing, although either explanation could be true. This rule is then combined with other rules for other measure deviations to increase the certainty that particular explanations are correct. Using this method of evaluating software measures provides a set of possible explanations describing a project's strengths and weaknesses. By using sets of rules in this manner, an automated system can examine the empirical evidence about a project and provide some insight into the project's status.

4.0 Using the SME

This section describes how the SME utilizes the concepts described above to provide its functionality. While the concepts of the SME are the most important aspect of the tool, understanding how to utilize those concepts to provide management support is also of interest. Attempting to build an SME-like tool requires knowledge of how to integrate the experience into a useful tool. The examples used are realistic in that they show the actual functionality of the SME, however, due to the inability to reproduce the color SME images, the graphics images are in black and white.

Comparison

One major function of the SME is the ability to observe data and compare it to models and previous development efforts. Figure 5, shows an example of using the SME to compare data to a model. In this example the manager is looking at the way error rate behaves on the project of interest. The current project is shown as the solid line and the model is shown as a band of what is considered "normal" for error rate. The x-axis shows the expected schedule for the project. That is, the start date and end date shown are the manager's estimates, however, the other phase dates shown are the expected phase dates for the project (as calculated by the SME). The tool

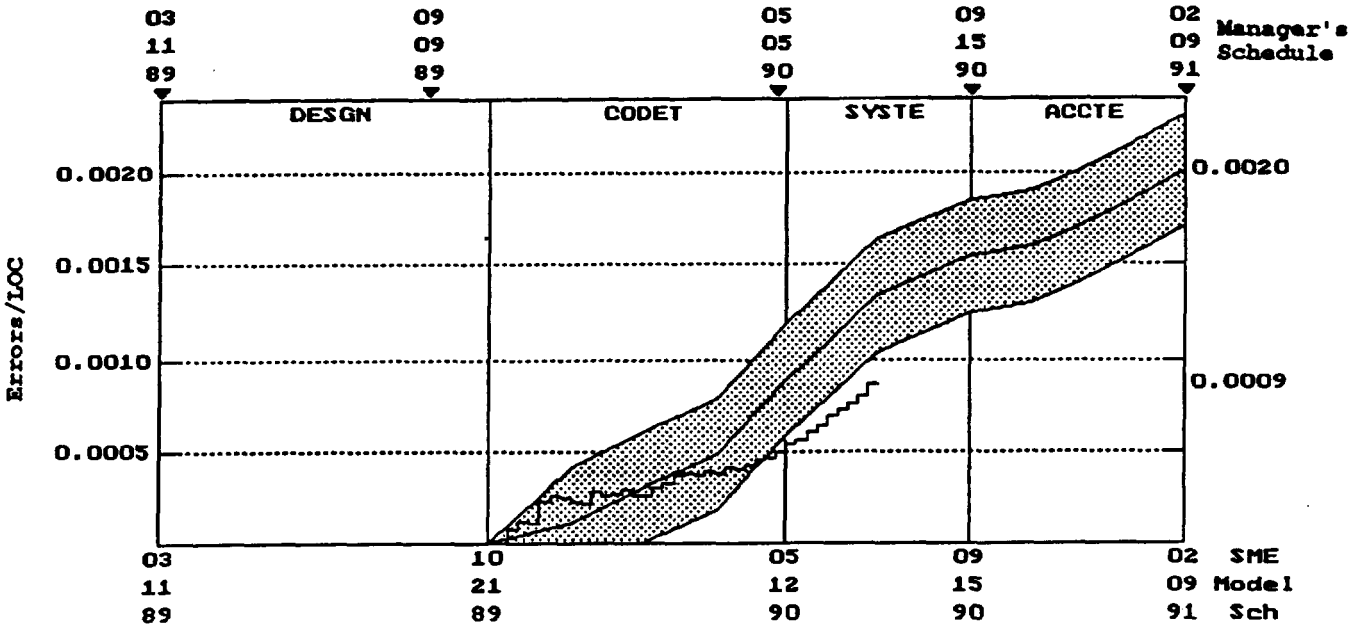


Figure 5: Rate of 'Reported Errors/Lines of Code' for Project A

also shows the manager's estimates for all the phase dates on the top of the screen. The Y-axis shows the error rate in errors per line of source code in the controlled library. Note that the phases represent a typical waterfall life cycle, with the major phases being design, code and unit test, system test, and acceptance test. By using this comparison, the manager is able to track such key items as error rate, productivity, and amount of computer time used. Additionally, the manager is able to overlay other projects' error rate patterns in order to compare the behavior of those projects to the current project.

Prediction

Figure 6, provides a look at another function of the SME. This figure is similar to the comparison figure, except that it also shows a predicted final value for the measure. In this figure, the measure of interest is computer use (in number of CPU hours). This is shown in absolute terms on the Y-axis. That is, the actual amount of time used on the machine is shown (it is not normalized). The SME allows the user to predict where the project will be when it is completed. This function utilizes the model and a projection of the progress of the project based on the measures in SME (eg. the project is 50% of the way through the code and test phase), to predict the final values of the measure, and of the schedule. In this example, the number of CPU hours on the project is predicted to be 1255, while the current estimate is 990 hours. Also, the project is predicted to take longer then the manager has estimated. Such predictions enable the software manager to gain another perspective on the final values of project measures and on the projected end date of the project.

Analysis

A key component of the SME is the utilization of expert systems technology for software management. Through experience, software managers are able to improve their ability to analyze software measurement data. Based on the measurement data and their experience, managers are able to identify the strengths and the weaknesses of a project. The SME utilizes a rule base

that captures managers' knowledge of how to perform such analysis. This rule base is then used to analyze deviations from the normal project. An example of such analysis is found in figure 7. In this figure, the error rate of the current project is lower then normal for this particular point in the development life cycle. The SME uses this information, information about other measures, and subjective data about the project to provide possible reasons for such a deviation. The top two explanations are then displayed for the user. In this case, the explanations are that insufficient testing is being performed and that an experienced development team is producing a superior project. Either of these two explanations might be correct, they only provide insight to the user as to possible explanations for the deviations. Other explanations are certainly possible; the user of the tool can obtain further data on why the system reached its conclusions and on the other conclusions. The user can also provide the system with more subjective information about the project of interest, perhaps leading to changes in the conclusions that are inferred.

Assessment

A final function of the SME is to utilize software measures to provide an assessment of the overall quality of a software project. An example of such an assessment is shown in figure 8. In this figure the bar graph shows the SME's rating of certain quality measures as they compare to the normal project in the environment at that point in its development. The quality factors shown are maintainability, reliability, and stability. Each of these factors can be determined by combining various software measurement data. For example, the quality factor of maintainability is calculated by adding the percentage of errors that are easy to isolate with the percentage of errors that are easy to correct. Thus, as these percentages increase the maintainability of the project is said to increase. For each quality factor displayed, SME has a specific definition for how to compute that factor. These definitions, which are really a form of a relationship, use a specific set of measures to compute the relative value of that quality indicator. Of course, SME also uses a model of how these factors behave over time in order to display the normal band on the graph. Quality assessment provides the software manager with an overall appraisal of how the project of

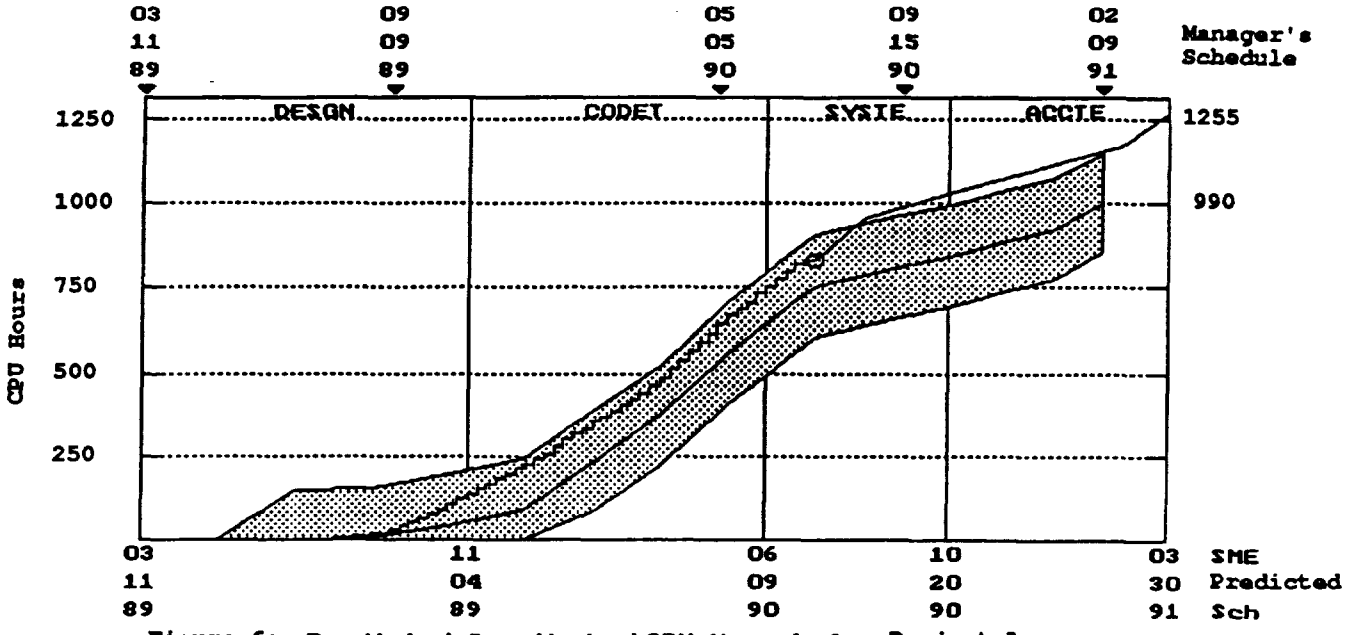


Figure 6: Predicted Growth in 'CPU Hours' for Project A

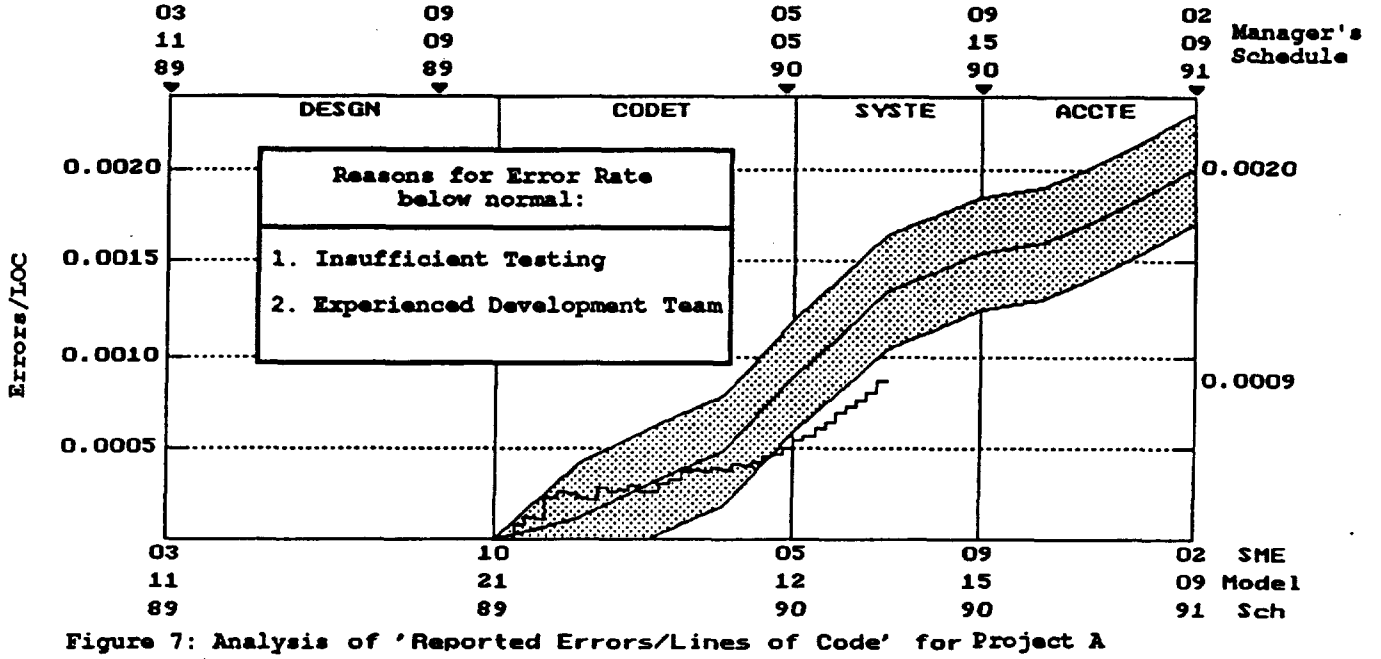


Figure 7: Analysis of 'Reported Errors/Lines of Code' for Project A

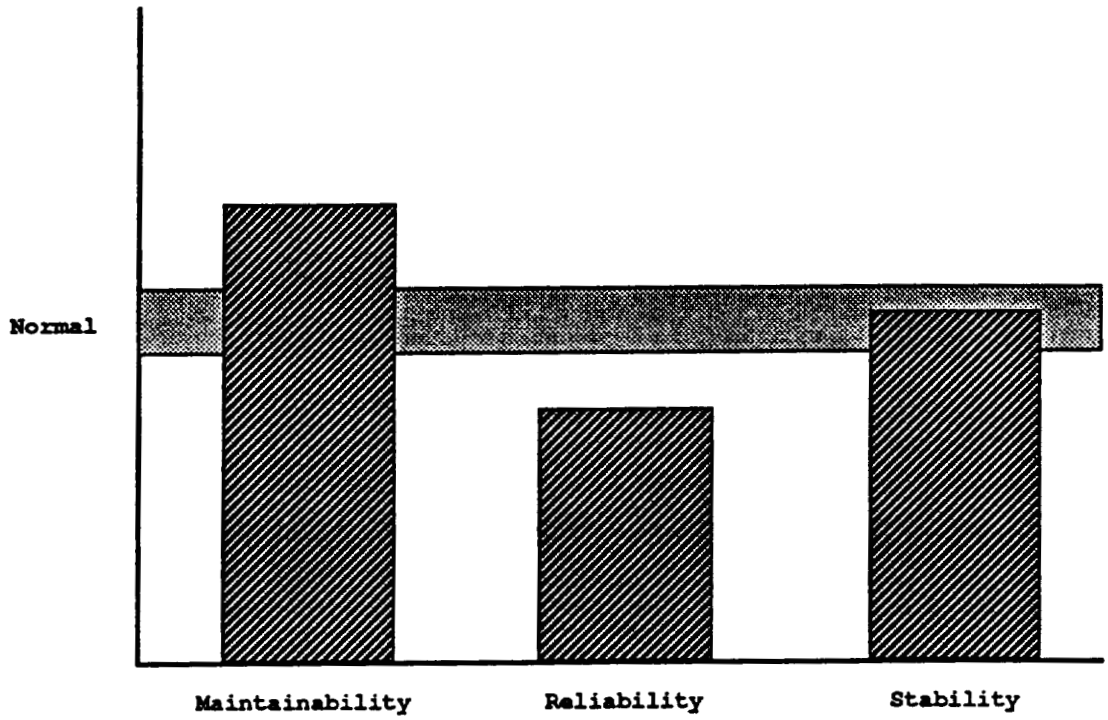


Figure 8: Overall Assessment Function

interest is doing compared to the normal quality measures in the environment.

5.0 SME as a Model Tool

Currently, the SME is being used by numerous software managers in the SEL software development environment to assist them in monitoring actual software projects. The SEL, as an experience factory [5], has provided the concepts necessary to build an SME for this particular software development domain. Other organizations can develop an SME-like tool by beginning to capture the experience of their environment. While within the SEL environment all three of the major components of SME have been well developed, other organizations may have only limited parts of the components. Such limitations should not be viewed as detrimental to the development of an SME. Similar tools should be developed using the experience available; they can then evolve into more complete tools as the local experience base provides additional artifacts for reuse.

The SME is an attempt to integrate a measurement process, the results of a longstanding software engineering research effort, and the expertise of software managers into a tool for managing and controlling software projects. As such, it provides for the utilization of corporate experience to manage ongoing software projects. An SME has been built for one particular software development organization. Other software development organizations should use the SME's concepts as a model for building similar tools for their environment. By providing the user with increased project awareness, predictions of key project parameters, expert analysis of software measures, and assessment of the overall quality of the development effort, an SME is extremely valuable to a software manager. Such a tool provides improved project management through the packaging of experience.

References

[1] Basili, V. R. and H. D. Rombach, "The TAME Project: Toward Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988, pp. 758-773.

[2] Selby, R. W., et al., "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development," Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society Press, May 1991, pp. 288-298.

[3] Kusumoto, S., et al., "GINGER: Data Collection and Analysis System," Technical Report, Osaka University, Osaka, Japan, June 1990,

[4] Valett, J., "The Dynamic Management Information Tool (Dynamite): Analysis of Prototype, Requirements, and Operational Scenarios," Master's Thesis, University of Maryland, May 1987.

[5] Basili, V.R., et al. "The Software Engineering Laboratory - An Operational Software Experience Factory," Proceedings of the 14th International Conference on Software Engineering, IEEE Computer Society Press, May 1992.

[6] So, M. et al., "SEL Data Base Organization and User's Guide (Revision 1)," SEL-89-101, The Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, Maryland, February 1990.

[7] Morusiewicz, L. and J. Valett, "Annotated Bibliography of Software Engineering Laboratory Literature," SEL-82-1006, The Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, Maryland, November 1991.

[8] Grady, R., "Work Product Analysis: The Philosopher's Stone of Software?," *IEEE Software*, March 1990, pp. 26-34.

[9] Decker, W., R. Hendrick, and J. Valett, "The Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules," SEL-91-001, The Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, Maryland, February 1991.

[10] Ramsey, C. and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989, pp. 747-759.

SECTION 4 – SOFTWARE MODELS

SECTION 4—SOFTWARE MODELS

The technical papers included in this section were originally prepared as indicated below.

- "The Software-Cycle Model for Re-Engineering and Reuse," J. W. Bailey and V. R. Basili, *Proceedings of the ACM Tri-Ada 91 Conference,* October 1991
- "On the Nature of Bias and Defects in the Software Specification Process," P. A. Straub and M. V. Zelkowitz, *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92),* September 1992
- "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," J. Tian, A. Porter, and M. V. Zelkowitz, *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92),* October 1992
- "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," L. C. Briand, V. R. Basili, and C. J. Hetmanski, *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92),* October 1992
- "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," L. C. Briand and V. R. Basili, *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92),* November 1992

The Software-Cycle Model for Re-Engineering and Reuse

John W. Bailey*
Victor R. Basili
The University of Maryland
Department of Computer Science
College Park, Maryland 20742

*also consultant with Rational, 6707 Democracy Blvd., Bethesda, Maryland 20817

Abstract

This paper reports on the progress of a study which will contribute to our ability to perform high-level, component-based programming by describing means to obtain useful components, methods for the configuration and integration of those components, and an underlying economic model of the costs and benefits associated with this approach to reuse. One goal of the study is to develop and demonstrate methods to recover reusable components from domain-specific software through a combination of tools, to perform the identification, extraction, and re-engineering of components, and domain experts, to direct the application of those tools. A second goal of the study is to enable the reuse of those components by identifying techniques for configuring and recombining the re-engineered software. This component-recovery or software-cycle model addresses not only the selection and re-engineering of components, but also their recombination into new programs. Once a model of reuse activities has been developed, the quantification of the costs and benefits of various reuse options will enable the development of an adaptable economic model of reuse, which is the principal goal of the overall study. This paper reports on the conception of the software-cycle model and on several supporting techniques of software recovery, measurement and reuse which will lead to the development of the desired economic model.

Motivation and Scope

Motivation for the development of an expert-assisted but highly structured and highly automatable model of software information capture and reuse stems in part from the recognition of the difficulty of using purely programming component-based approaches to reuse libraries. For certain kinds of objects and components a strict programming component-based library is adequate. The success of object-oriented and object-based approaches have been the most notable in this regard. However, the inability for such libraries to capture a sufficient amount knowledge to dramatically reduce subsequent software development costs in a general and problem-independent way has also been observed. On the other hand, models of software reuse which utilize domain experts in pervasive and undirected ways are also unlikely to provide a complete solution due to the large amount of responsibility and effort which is centralized in the contribution of such experts. The present work provides a structured model of information identification and reuse which is both feasible and suitable for further development and refinement.

Using the Ada language, this paper provides examples of techniques for choosing, re-engineering, and recombining components into programs. It also describes rudimentary methods for quantifying the effort to extract reusable components from existing programs as well as the effort to recombine them into new programs. It does not include the cataloging and retrieval of components, nor does it include a mechanism to quantify reusability based on empirically-derived frequency-of-use measures. It does model a proposed cycle of software development, use, re-engineering, and reuse, but it does not attempt to model other aspects of reuse within a software development environment, such as pure knowledge and experience. Other recent research papers and technical reports have covered this larger scope [Basili and Rombach], [Basili and Caldiera].

Introduction

Any component of software is seen to be composed of many functional and declarative details, some of which pertain to the specific problem being solved by the program containing that component, some of which pertain to the general application domain of the containing program, and some of which pertain to neither the problem nor the

domain, but rather define the essence of the component's function in the abstract. Therefore, to direct the selection and re-engineering of components of software, three levels of functional specificity of the software which constitutes any component are defined: 1) problem-specific details which would be likely to differ between this and another similar application in the same domain, 2) domain-specific details which are not likely to differ between this and another similar application in the same domain but which would be unlikely to be appropriate outside of this domain, and 3) essential aspects which comprise the abstract functional core of the component and without which the component would be meaningless.

The three levels cannot be absolutely defined, nor can a given detail be deterministically assigned to a level, since from different points of view, a given detail could be thought of as belonging to different levels of specificity. Two analyses of a given component could possibly identify different sets of details at each of the three levels. However, an analysis of a candidate component for the purpose of directing the re-engineering and reuse processes must assign each identifiable detail to one of the three levels.

Once specificity levels have been assigned to all details of a candidate component, a measurement of the effort required to remove each of the problem-specific details is obtained in order to estimate the total effort to generalize the component for reuse within its domain. Further, a measurement of the effort required to remove each of the domain-specific details is obtained in order to estimate the total effort to generalize the component for reuse in other domains. If these measurements show the cost-effectiveness of either of these generalizations, then the candidate component is suitably generalized and placed in either a domain-specific or domain-independent repository, as is appropriate.

In order to assign specificity levels to all the constituent details of a candidate component, domain experts may have to be consulted. However, automation to support the identification of the details and to support the component generalization through their removal can be used to streamline the process. Further, there may be ways to capture the domain experts' decisions and the reasons for them, in order to partially automate or support any subsequent decision making which follows similar patterns.

To support the generalization process and its quantification, three styles of software component reuse which are currently being practiced are identified and examined for their adaptability to the model. These reuse styles are termed *layered*, *tailored*, and *generated* reuse. Examples illustrating them, and demonstrating how they are related by an underlying dimension of generality, are shown.

Along with these examples, proposals are given for how to measure the amount of re-engineering required to derive components suitable for the different methods of reuse, as well as the amount of effort required to recombine components using the different methods. As effort is expended to make a component more general, more opportunities to reuse it become available. However, each of those reuse opportunities will have to resupply the specifics required for the reusable component to perform its function in the new context, implying an amount of reuse effort which is proportional to the degree of generality of the component.

Therefore, an economic equation presents itself, which is how to optimize the sometimes competing factors of generalization effort, reuse effort, and breadth of utility. The solution to this equation will have to wait until more work is done on the probability of reuse for a given generalization, and other factors. Rather hard questions figure in to this equation, such as the cost-benefit of constraining a solution to take advantage of an available component (which amounts to establishing and following standards) as opposed to developing a more suitable one, and even the cost of classifying, storing and retrieving components. Developing a framework for an economic model which captures these factors is the first step to a greater understanding of these issues. The last section relates the activities defined in the software-cycle process model to this economic model of reuse.

The Software-Cycle Model

This section describes the model of software development which underlies this study. The model proposes the recycling of existing software into components which can be combined into new programs. This proposed *software cycle* takes place in the context of a software development organization and allows effort already applied to the creation of previous programs to be recaptured and used to reduce the effort needed to create new programs. This software-cycle model is consistent with models of experience capture and flow within a development organization as described by [Basili and Rombach] and [Basili and Caldiera]. It describes in detail, and proposes an implementation for, one aspect of the more comprehensive experience factory described in those studies.

The software-cycle model is so-named to describe the flow of information and experience, in the form of software, into newly developed programs where it can be recovered and packaged for efficient reuse in subsequently developed software programs. The capture and reuse of information at the delivery point of the conventional software lifecycle is clearly not the only time at which such information is

accessible. However, this approach is chosen because at the time that software is delivered, the information is packaged in a concrete form (software programs) which can be analyzed and manipulated. Also, a substantial amount of information may be available from previously-developed programs which is not recorded in any form other than the delivered software. Further, by instituting an approach which applies effort to capture reusable information at this stage, the software development organization has the choice to separate the information recovery and repackaging from the effort to develop the software, and to conduct those activities independently and in parallel. So, for pragmatic reasons, the present model of information flow in a software development organization uses developed software as the main source for recoverable information. (See also [Caldiera and Basili].)

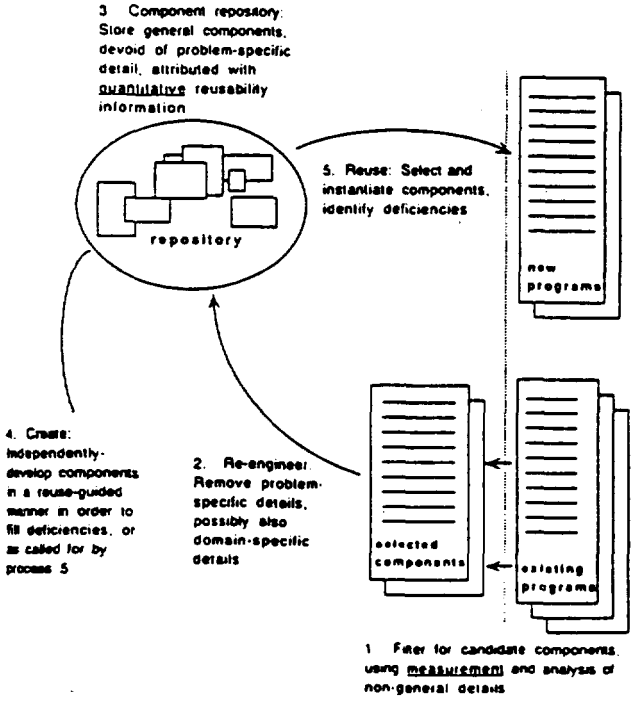


Figure 1 The processes involved in the Software Cycle Model

As shown in Figure 1, existing programs are examined for candidate reusable components. For the purpose of this study, a component can be any definable portion of software. Obvious examples are individual, or sets of, subroutines, subprograms, functions, paragraphs, packages, or other structuring features of the software language in use. A re-engineered component can be any of these, although it can also be nothing more than a template or a set of instructions for a software generation routine.

A re-engineered component can be intended either for reuse only within a particular domain or reuse across many domains. If a component is only intended for reuse within a domain, its re-engineering seeks to remove any problem-specific details from it, but to allow any domain-specific details to remain. Such components are termed *domain-specific components*. If a component is intended for reuse across domains, however, then its re-engineering would attempt to remove all domain-specific details as well as the problem-specific details, leaving only essential function. This kind of component is termed a *domain-independent component*. Leaving a component insufficiently general to be used across domains obviously limits the number of opportunities it might enjoy for reuse. However, there are significant compensating advantages. A domain-specific component retains more details which then do not have to be resupplied by the reuse client. Also, the generalization effort to reach only problem-independence is usually less than the generalization effort required to reach domain-independence. So, by accepting a constrained reuse scope, a component can be easier to generalize as well as easier to reuse.

A candidate component for re-engineering is one which has identifiable problem-specific or domain-specific details and which can be feasibly re-engineered to eliminate the presence of some or all of those details. A domain expert may be needed to differentiate between problem-specific and domain-specific details, and measurement of the estimated generalization effort is needed to determine the feasibility of the re-engineering. Some components may be candidates to yield a domain-specific component after re-engineering but not a domain-independent component. Other components may be candidates to yield domain-independent components (possibly in addition to domain-specific components), while still others may not be good candidates to yield either category of reusable component.

The goal of reuse re-engineering is to be able to isolate and then to replace the problem-specific and/or the domain-specific aspects of a component so that it can be made to operate in different contexts. A component might be viewed as a blend of general function, which defines its essence, and specific function which relates to the current context or declarations on which the general function is performed. This is shown graphically in Figure 2a. The general function, shown in light grey, is that which is essential to the component or that which defines the nature of the component. The specific function, shown in dark grey, can either be problem-specific or domain-specific. As mentioned, it may be necessary to consult domain experts to distinguish between a problem-specific detail and a domain-specific detail. However, given a sufficient body of experience, it may be possible to predict the specificity of a detail via a predictive function that is tailored by previous expert decisions, or by statistical

analyses of several similar components in the same domain.

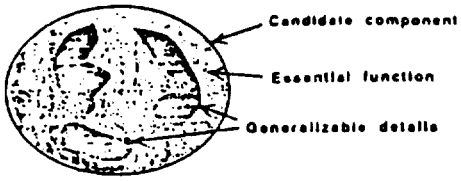


Figure 2a.

Typical candidate component, showing that it is a combination of essential function which defines the component and problem-specific or domain-specific details which can potentially be generalized in order to re-engineer the component into a more reusable one.

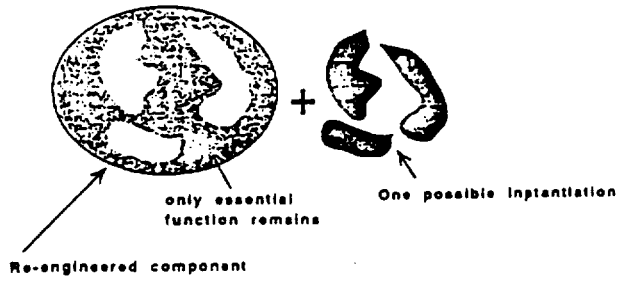


Figure 2b.

After re-engineering, the essential functionality remains in the reusable component but problem-specific or context-specific details are eliminated and become the responsibility of the reuser to provide. One possible instantiation could result in the original component again, but many other instantiations are now possible

Figure 2b shows an imaginary candidate component which contains both essential function, which is general, and specific details which, if altered, could allow the component to contribute its functionality in different contexts. These specific details, shown in dark grey, have been removed from the body of the component to signify that they are now viewed as only one of potentially many possible instantiations of the remaining, general component. The re-engineering process of the software-cycle model seeks to locate and remove these non-general aspects (either only the problem-specific aspects or, possibly, the domain-specific aspects as well) and to relegate them to the responsibility of the reuser as part of the component's instantiation. The techniques for the removal of these details are discussed as part of the section on re-engineering techniques which follows. It will be shown there that the re-engineered component does not need to be expressed in the programming language of the original candidate component which was used to produce it. It might be a pre-processable component or a component generator which can be used to produce components when necessary. In these cases, it is the template or the generator that is reusable, since any subsequently required components would be produced on demand and would not, themselves, be considered reusable.

Separated and re-engineered (generalized) components are stored in a repository to be made available to the developers of new software. Similar to the process of consulting domain experts when categorizing the details which need to be generalized out of candidate components, repository experts may have to be consulted to assist in the location and instantiation of required components in the repository. Repository experts could possibly choose from among various schemes to satisfy the needs of a developer. Certain choices might provide more utility but might come with more restrictions or limitations of options. Also, the repository expert might choose from different methods to arrive at functionally the same result to the requesting developer, for example by either generating the software or by providing a tailorable component.

Components in the repository are attributed with measurement information describing the expected effort to instantiate them for reuse. In many cases, this instantiation becomes the responsibility of the reusing developer, for example when the component is already a structural component in the developer's language of choice and simply must be supplied with actual parameters to serve the developer's need. In other cases, the instantiation can be the responsibility of the repository expert, who might have to produce components for the developer from templates, rules, instance specifications, and generator programs. In either case, the measurement attribute of a component will guide its users when deciding whether to select it or not, and how much effort to expect to expend configuring it for reuse.

A request for software components might be unfillable given the current state of a repository. In this case, the repository experts can work with the developer to design and create a new component which will not only serve the current need but which will become an instant candidate for insertion into the repository, with a minimum of re-engineering. Or, gaps in the capabilities of the repository can be identified by the experts prior to a specific need, and special developments can be guided, specifically for the purpose of supplying components to fill those gaps. In the software-cycle model, any new development is done with reuse in mind, specifically with an eye toward further populating the component repository.

Neither of these last two topics, the selection of components from a repository and the direct development of components rather than through re-engineering, are currently part of the study. They are mentioned here in order to complete the software cycle depicted in Figure 1.

The major emphases of the study are the identification of candidate reusable components from among existing software, the re-engineering of those components to improve their generality, the measurement of those processes, and the development of an economic model which can assist an organization in optimizing its software cycle costs.

Reuse Modes and Methods

By studying the dependencies among software elements, a determination can be made of the reusability of those elements in other contexts. For example, if a component of a program uses or depends upon another component, then the first component would not normally be reusable in another program where the second component was not also present. On the other hand, a component of a software program which does not depend on any other software can be reused in any context (ignoring for the moment whether or not it performs any useful purpose in that context). The issue of software independence is at the heart of this study.

It will be seen that increased independence of a software component often comes at the cost of functionality. The ideal software reuse re-engineering process would provide a means of preserving all of the function or utility of a component while also making it independent of problem-specific or domain-specific details. However, this is not possible in most cases since some of the desired functionality is likely to be captured by those specific details, and removing the details will remove that functionality. This study describes a compromise solution, which is first to generalize a component, and then to systematize the means to configure it in order to restore the specific function required in a particular context of reuse.

A scheme to maintain generalized, reusable components in a repository, in addition to a means of configuring them in different ways for different domains or contexts, enables a repository with a manageable number of components to be described. Without the ability to instantiate a given component in different ways for different usages, a repository would have to contain many times as many assets in order to serve the same need. In order to avoid this problem, this work recommends storing fewer components, each of which is sufficiently general to be able to operate in various contexts, and then providing methods to instantiate them to provide functionality in those contexts.

By examining existing successes in software reuse, it can be seen that there are three different but related ways of making software components which are general and independent, and yet which remain capable of being instantiated with problem-specific details. An important premise of this work is that software which is general in these ways does not necessarily need to be developed directly. Instead, it is often possible to re-engineer existing software so that it achieves the necessary independence.

For this study, the three modes are termed *layered*, *tailored*, and *generated*. Each mode describes components which can be combined to develop larger programs. However, a tailored component can be made more flexible and general than a layered component and a generated component can be the most flexible and general of all. On the other hand, a layered component is the easiest to reuse, requiring the least effort on the part of the client to incorporate it into a program, while a generated component is the most difficult to reuse.

What all of these techniques strive for is the absence of dependence from the reused software on external declarations, which would hamper the generality of the software. In other words, a component of reusable software should ideally not be expected to "know" about declarations and other components which are problem-specific. A reusable resource which requires the reuser to also include other common denominator components, which contain needed declarations, is not as reusable as one which has no such requirements.

Within the confines of a single domain, however, certain dependencies can be tolerated, since the users can be expected to guarantee the minimum required declaration space across all occurrences of reuse of a component. This result opens up vast new ranges of possibilities, since the generality of a component need no longer be absolute but rather need only be general with respect to a certain domain or domains. No expectation of generality within other domains is maintained. Domain-specific reusability implies a certain amount of built-in dependence whereas wide-scale reusability or generality precludes this possibility. By allowing domain-specific constraints, the possibilities for identifying reusable components expand enormously but the breadth of applicability for each component is limited to that domain.

Layered Reuse

Layered reuse is used to describe the case where reusable functions or operations are viewed simply as abstract primitives which are callable from within the language of the client. A math library, probably the most commonly cited example of reuse, and one which is often viewed as an ideal, is an example of layered reuse. Analogous to a math package, other common examples are packages of utilities which operate on universal types or concepts, such as string handling utilities and time utilities. Other successes in layered software reuse include user interface

or I/O toolkits, graphical display toolkits, runtime kernels, and layered network protocol software.

Layered reusability is often viewed as the goal for a library of reusable components, where a sufficiently rich set of abstract operations would be available to an applications programmer in order to minimize the effort required to generate a new system. In addition to the previously mentioned independence from other components, an additional recommendation for the success of a layered component is that the data on its interface be expressed in terms of standard types. This restriction allows the client software to communicate with the reusable component without the additional complexity of adhering to specific non-standard types. One reason that a math library is so inherently reusable, for instance, is that real numbers are a universal way of expressing the values used by and returned by the mathematical functions in a library. Any language which supports real numbers can make available a corresponding set of mathematical functions.

However, unlike the portability enjoyed when restricting one's domain to a universal concept such as real numbers, a considerable amount of software which might otherwise be available for reuse is written to operate on problem-specific types and data structures. This is the case whether those types are named and declared as in Pascal or Smalltalk, are common data areas as in Fortran, or are merely locations in memory as in assembly language. Components can still be written in a layered manner but in these cases they typically depend so heavily on specific data structures that they are limited to being reused only where identical data structures or other operands are present. It is not always possible to parameterize a component with respect to all of its assumptions about context. Because of these limitations on the applicability of a layered component, constructing comprehensive reusable libraries of them in languages such as Ada has been harder than might have been expected.

Tailored Reuse

Another category of successful reuse is tailored reuse, where configuration of the reusable software is required in order to allow it to interoperate properly with the client software. A familiar example of such reuse is seen with database management systems which require tailoring in order to handle records of the user-defined structures. Simpler examples of tailored reuse are generic data structures which allow the client software to create stacks, queues, lists, etc., of application-specific types or to search through or sort objects of those types. Still other examples of tailored reuse are forms management systems which are customized by parameterization, expert systems which must be initialized with rules, spreadsheets which must be supplied with formulas, and statistics packages which must be provided with data sets and programs to achieve the desired results.

Tailoring in this way is accomplished before the component is called, but it happens automatically at execution time as part of the language behavior. Whereas in layered reuse a client simply calls a component with the proper parameters, tailored reuse implies a two-step process where a component is first molded to the specific configuration required by the current context and is then called to perform its function.

The generic feature of Ada allows certain kinds of tailoring, in the form of generic parameterization, to be accomplished. Because of the static checking enforced by Ada, however, only a limited amount of parameterizations are possible. Other languages have different mechanisms for accomplishing this parameterization. Most notably, assembly languages employ very flexible macro expansions which can be quite powerful. However, object-oriented languages have traditionally used a more flexible form of layering (full inheritance) while overlooking the possibility for component parameterization. (Future revisions to C++, however, are expected to include a template mechanism to allow within-language tailoring [Ellis and Stroustrup].)

Generated Reuse

The third category of reuse, generated reuse, occurs when the reusable software is used as a generator program rather than being incorporated directly into the final application. The required software is emitted as a result of the generator program operating on input tables or files. Typically, only the generator and not the generated software is reused. The generated software is regenerated, as opposed to being modified directly, if changes are required. Whereas layered and tailored reuse take advantage of language-supported features (subprograms and generics in the case of the Ada language) generated reuse requires additional tooling to accomplish a kind of tailoring which is external to the implementation language.

A common example of generated reuse, which perhaps stretches the definition somewhat, is a compiler, which accepts files of a high-order language and emits software in a machine-executable form. One reason that it may seem unconventional to think of a compiler as reusable software is that its output is not directly manipulated or even observed by the compiler's users. Nevertheless, it fits the definition here for generated reuse (which could be thought of as a batch form of tailored reuse).

Other common examples, where the generated output is more likely to be manipulated or at least observed by the

users of the generator, are fourth-generation languages, user interface generators, test case generators, parser generators and table-driven forms management systems. At least one large Ada development is making substantial use of generated reuse in an MIS system development, through the use of a specially-developed generator [AIC].

Table 1 is a summary of the modes of software reuse described and the examples mentioned for each.

Layered:
- Math libraries
- Common utilities packages
- User interface or I/O toolkits
- Graphics kernel systems
- Runtime kernels
- Network layered software

Tailored:
- Database management systems
- Forms management systems (runtime configured)
- Expert systems
- Spreadsheets
- Statistics packages
- Generic data structures

Generated:
- Forms management systems (file driven)
- User interface generators
- Test-case generators
- High-order languages
- Fourth-generation languages
- Parser generators
- MIS systems

Table 1. Reuse Modes and Examples

The distinctions between these categories can sometimes become blurred. For example, whether a reusable package is configured at run time by parameterization (tailored) or in advance by tables such that it emits a separate program (generated) may not be of any real consequence. In fact, the examples given in one category often have analogs which exist in the other category. For example, forms management systems already exist in both generated and tailored versions. Although parser generators are typically generated components, since they are stand-alone grammar-driven programs which emit desired software, they could instead be incorporated into the end-product and re-emit their parsers on the fly. The obvious reason not to do this is for efficiency of repeated use of the same output. However, an interpreter for a language can be thought of as a compiler which is configured to perform as tailorable software. In this case, the run-time efficiency is traded off for the flexibility of being able to alter the "parameterization" (the interpreted program) quickly and easily.

A Simple Example

As a simple example of how a low-level component can be viewed as a generalizable layer of function, consider the following error-reporting routine.

```
with Text_Io;
procedure Gyro_Speed_Error is
begin
  Text_Io.Put_Line ("Error: The gyros are not up to speed.");
end Gyro_Speed_Error;
```

This highly specific routine represents one end of the generality scale. It is easy to use, requiring a simple parameterless call, but might not be likely to be widely called upon within a program. There are three observable details within this unit: 1) the use of Text_Io.Put_Line to report the error message, 2) the use of the standard output device to display the error, and 3) the choice of the literal string to be displayed.

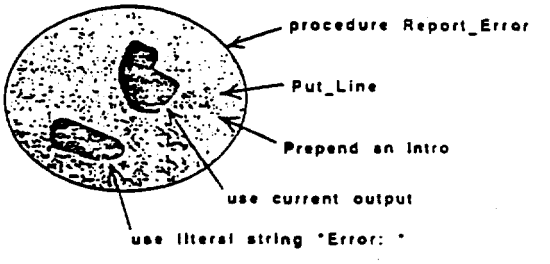


Figure 3a.

In the example from the text, procedure Report_Error was seen to be composed of four decisions. Two are considered part of the essential functionality and two are considered to be problem-specific details

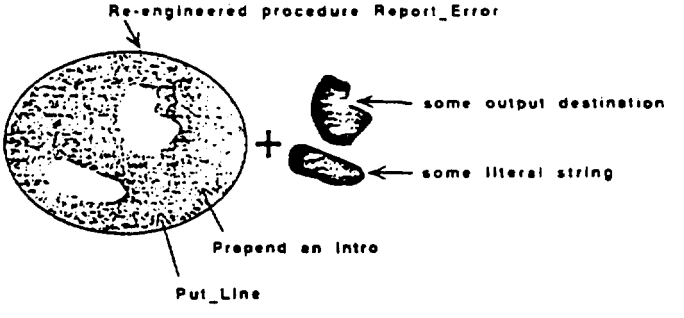


Figure 3b.

The re-engineered version of Report_Error shows the two problem-specific details removed from the component, to be supplied by the *re-user*. The *intrinsic functional* aspects of the component remain. Other interpretations of the re-engineering decisions to be applied could possibly remove one of these, as well.

A consultation with a domain expert might result in our choice to parameterize the exact error message to be reported, which might yield the more sensible reporting routine, shown below.

```
with Text_Io;
procedure Report_Error (Message : String) is
begin
  Text_Io.Put_Line ("Error: " & Message);
end Report_Error;
```

This version of the unit is depicted in Figure 3a. Had we performed the transformation without expert consultation we might have simply parameterized the entire message. However, in our hypothetical problem domain we will assume that the expert recommended retaining a hard-coded standard prefix in order to facilitate the post-processing of the log file. Also, this generalization has cost us the part of the original functionality which spelled out the exact error message. Since the client must now supply this string, we have increased the effort to use the unit by making it more general.

The generalization of a value (a string value in this case) is the easiest kind of transformation since it can be performed with a simple value parameter. Since the parameter type is language-defined (type String) there is no further complexity to exposing this parameter in the procedure interface. Also, the effort to configure the component amounts to simply defining the error message string as a parameter. Again, this kind of reuse is the easiest.

The procedure above still assumes that the user intends the message to be written to the current output device using Put_Line. That constitutes part of the retained functionality of this component. In the process, we have also added the detail that the standard prefix "Error: " will always appear.

Additional consultation with a domain expert might reveal that the assumed use of the standard output device is another problem-specific detail. A later reuser of this component who was working on a different problem in the same domain might not want to be bound by that assumption. Again, Ada provides a simple way to parameterize the component so that users can specify the output device. Again, however, this generalization comes at the cost of functionality. In this case, the functionality which is lost is the assumption is that the current output device is to be used. Default parameters can sometimes provide an opportunity to restore such assumptions while retaining the generality, as will be shown later. The parameterized version of the unit which follows removes the assumption of using the current output device but retains the function of writing the literal string "Error: " followed by the caller's message.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         On_Device : Text_Io.File_Type) is
begin
  Text_Io.Put_Line (On_Device, "Error: " & Message);
end Report_Error;
```

Notice that the user is now required to do additional work. Instead of simply providing the error message, the desired output device or file must be provided. That decision has shifted from the component to the (re)user. Again, this is a form of value parameterization, the easiest form of both generalization and reuse configuration.

An additional part of the functionality of the component is the literal string prepended to the caller's message. As shown below, this could also be parameterized, again removing that specific functionality but generalizing the component on that behavior. This requires yet one more piece of information from the user as part of the information needed for this component to perform its work, however once again it is a low-cost value parameterization.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         Intro : String;
         On_Device : Text_Io.File_Type) is
begin
  Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

This generalized component is depicted in Figure 3b. This might constitute a domain-independent version of the reporting routine, according to our domain experts, although the only way to be certain that a component is compatible with all domains is to ensure that it does not depend on any other components. In Ada any such dependencies are revealed by the context clause. A later transformation will eliminate the dependence on Text_Io.

As noted, Ada affords us an opportunity to restore the assumption of using the specific string "Error: " and the standard output device through the use of default parameters without reducing the generality. This is shown below.

```
with Text_Io;
procedure Report_Error
        (Message : String;
         Intro : String := "Error: ";
         On_Device : Text_Io.File_Type :=
            Text_Io.Standard_Output) is
begin
  Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

At this point, two details remain (the use of

Text_Io.Put_Line and the prepending of a user string). The use of Put_Line could be removed through tailoring (below) but the removal of the choice to concatenate an introductory string could not be done within the language. For that degree of flexibility, generated reuse would be required. Once a generalization is needed which is not language-supported, the costs are considerably higher. One way to reduce those costs is to provide tool support for the generalization, a process which amounts to establishing a new language to accomplish the generlization. The MIS system described in [AIC] has reduced their software generation costs in this fashion.

This points out the obvious conclusion that the cost of a generalization depends on the level of language or tool support for it. One way to estimate cost is to begin with an ordinal scale of difficulty and then to move to a more detailed scale after more analysis has been done. For example, it was noted that value parameterization is relatively straightforward. This would be at the lowest end of an ordinal effort scale. Above that would be tailoring parameterization such as Ada's generic formal type and subprogram parameters. At the hardest end of the scale would be software generation, with tool-supported generation being easier than custom-built generation. A more detailed approach to effort would be to relate the cost to the number of lines of code that must be written, changed, or added.

It can require a judgment call to choose what details to remove and what function to leave in the component. For example, in the above example, the fact that the original literal string was broken up into a standard prefix and a user-supplied message was only one possibility for generalization. One guideline is to leave operational parts of a component intact and to allow the operands to be supplied by the reuser. A discussion of the separation of operations from operands can be found in [Bailey and Basili].

The simple error-reporting example from before can also be re-engineered into a tailored component using the Ada language. The difference between this result and the layered result is that the reusers will have to perform slightly more work in order to instantiate the component, but then subsequent calls can be simpler. As suggested, tailoring in Ada through the use of generics is seen as a harder process than value parameterization but easier than software generation. A tailored example of the component follows.

```
with Text_Io;
generic
   Intro : String := "Error: ";
   On_Device : Text_Io.File_Type := Text_Io.Current_Output;
procedure Report_Error (Message : String);

procedure Report_Error (Message : String) is
begin
   Text_Io.Put_Line (On_Device, Intro & Message);
end Report_Error;
```

Unfortunately, this is illegal in Ada since a limited type (Text_Io.File_Type) is not permitted as a generic value parameter. This is an example of where strong static checking can be at cross purposes with generalization and reuse. If it were legal, nevertheless, the user would have the responsibility for providing the introductory string and the output device one time (at the time of the generic instantiation) thus tailoring the component for further reuse. From then on, the component would be no more difficult to use (from the standpoint of parameterization) than the original non-general version.

To avoid this limitation of generic parameters, a solution could be obtained by generating the specific component desired, using tools outside of the Ada language. The generated component could look exactly like the original component but the reusable software would no longer be considered the component itself, but rather the generator which creates it. In this case, the generator would emit a Report_Error procedure which was hard-coded to write the error message on a given device. The value of that device would be given as a parameter to the generator. More examples of generation are shown later.

A different tailoring would also be possible. As mentioned earlier, the dependence on Text_Io can be eliminated by requiring that the client tailor the component to use a particular string-processing routine. This makes the component completely independent, with the persistence of the use of a standard prefix as the only detail which is retained from the original version.

```
generic
   Intro : String := "Error: ";
   with procedure Put (S : String);
procedure Report_Error (Message : String);

procedure Report_Error (Message : String) is
begin
   Put (Intro & Message);
end Report_Error;
```

Note that this most general version is also the least functional. Nevertheless, the ability to tailor the component once within a program and to then use it with the same level of effort as the first layered transformation makes it of some value. The reuser has additional work to do with this solution, as well. For example, unless the error messages are to be written to standard output, the subprogram to be passed to the generic formal Put

procedure has to be written. This means that the effort to reuse a tailored component could be greater than the effort to reuse a component generator. So, the effort to generalize is not always proportional to the corresponding effort to reuse.

By examining existing systems and by observing the opportunities to generalize their parts according to these different methods of reuse, choices become available in the ways in which the software can be re-engineered for future reuse. The next section describes a simple mail system in terms of its conventional configuration as a custom-built application and then in terms of the various ways the parts of it can be generalized using the above methods.

Re-Engineering a Simple Electronic Mail System

This section takes a simple electronic mail system through transformations to yield components which can be combined using the three methods described above. In the interests of space, parts of the examples and some identifier names have been abbreviated, and no bodies are shown. Complete listings of the examples are available from the authors.

In a conventional design, one component, or package, of a mail system could be used to manage the mailboxes of the users and a second could manage the messages, or the constituents of a mailbox. This would represent a conventional encapsulated or "object-based" design of the system where the mailbox package would allow operations such as create, add a message, delete a message, return a message, and perhaps displaying a directory of messages, maintaining the status of each message, and so on. The message package would allow message creation and display, and possibly reply construction, forwarding, etc.

In a typical arrangement, using either Ada or an object-oriented language such as Smalltalk, the mailbox package (or object) would depend upon the message package to obtain the use of the declaration of message objects, in order to arrange those objects into mailboxes. In Ada, the specifications for each of these two packages might reasonably be:

```
package Messages is
   type Username is ...
   type Line is ...
   type Text is ...
   type Message is private;
   procedure Set_Sender (M : in out Message; To : Username);
   procedure Set_Receiver (M : in out Message; To : Username);
   procedure Set_Subject (M : in out Message; To : Line);
   procedure Set_Body (M : in out Message; To : Text);
   function Sender_Of (Msg : Message) return Username;
   function Receiver_Of (Msg : Message) return Username;
   function Subject_Of (Msg : Message) return Line;
   function Body_Of (Msg : Message) return Text;
private
   type Message is
      record
         Sender : Username;
         Receiver : Username;
         Subject : Line;
         Msg_Body : Text;
      end record;
end Messages;

with Messages;
package Mailboxes is
   type Message is new Messages.Message;
   -- derive an equivalent type Message
   Max_Mailbox_Size : Natural := 1000;
   subtype Box_Size is Natural range 0 .. Max_Mailbox_Size;
   type Mailbox (Size : Box_Size := 0) is private;
   procedure Store (Box : Mailbox; Owner : String);
   procedure Retrieve (Box : in out Mailbox; Owner : String);
   function Size (Of_Box : Mailbox) return Box_Size;
   function Msg_At (Position : Natural; In_Box : Mailbox)
      return Message;
   procedure Remove (Num : Positive; In_Box : in out Mailbox);
   procedure Append (Msg : Message; To_Box : in out Mailbox);
   procedure Mark_Read (N : Natural; In_Box : in out Mailbox);
   procedure Mark_Unread  ...
   procedure Mark_Answered ...
   procedure Mark_Deleted  ...
   procedure Mark_Undeleted ...
   function Is_Read
      (Msg_Number : Natural; In_Box : Mailbox) return Boolean;
   function Is_Answered ...
   function Is_Deleted  ...
   No_Msg_At_Position : exception;
private
   type Attributes is (Deleted, Read, Answered);
   type Attr_Sets is array (Attributes) of Boolean;
   type Mail_Item is
      record
         Item : Message;
         Status : Attr_Sets;
      end record;
   type Item_Array is array (Positive range <>) of Mail_Item;
   type Mailbox (Size : Box_Size := 0) is
      record
         Items : Item_Array (1 .. Size);
      end record;
end Mailboxes;
```

These packages are depicted in Figure 4a. As shown, the Messages package is an example of an independently reusable layer, and the Mailboxes package constitutes a layer on top of the Messages package. (Since the constituent types of Username, Line, and Text are not shown, it might be the case that they would be comprised of user-defined types, making the Messages package dependent on other client software.) Realizing that the decision of how to implement the constituents of a message represents one of the opportunities for generalization of this package, the components of a message could be supplied as parameters to a generic version of this package. This would constitute a tailored version of the package:

```
generic
  type Username is private;
  type Line is private;
  type Text is private;
package Gen_Messages is
  type Message is private;
  ... -- as before
end Gen_Messages;
```

This generalization is shown in the top part of Figure 4b. The effort to perform this tailored generalization is in line with other tailoring efforts discussed in the previous section. The declaration of three generic formal parameters is one measure of the work performed. Also, the reuse effort implies the declaration of actual type parameters to be associated with these generic formal types. One way to quantify the effort to generalize, then, is to claim that three declarations are required. Three declarations are also required of the client reuser.

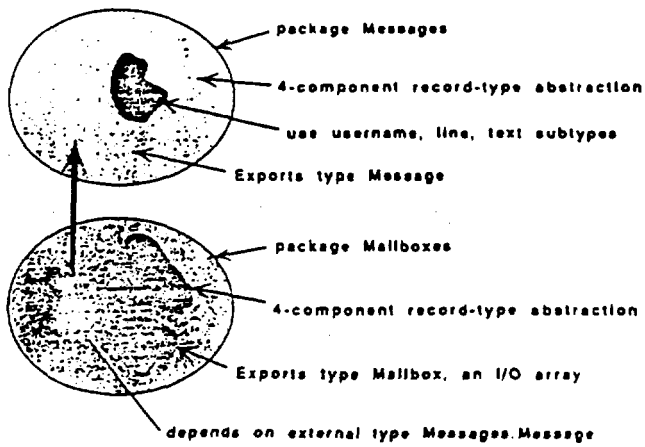


Figure 4a.

...ing the conventions shown previously, this depicts the process of tailoring the Messages and Mailboxes packages from the text.

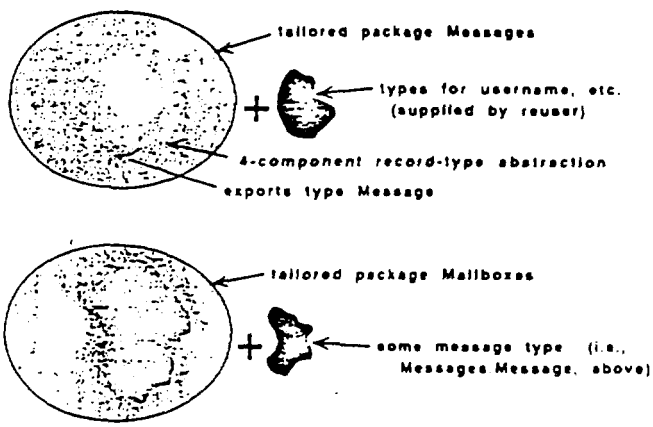


Figure 4b

...e specific component types of a Message have been removed as well as the dependency of Mailboxes on Messages. The reuser will re-establish this link.

Going beyond this somewhat tailored version, notice that even the structure of a message could be a candidate generalization. In this case, tailoring would be difficult or impossible within the confines of the Ada language so generation is required. Generation is feasible since the contents of the Messages package could be deterministically described if one were to specify the constituent components of a message. For example, if no subject line were wanted, the original package could instead have been written:

```
package Messages is
  type Username is ...
  type Text is ...
  type Message is private;
  -- procedures Set_Sender, Set_Receiver, Set_Body
  -- functions Sender_Of, Receiver_Of, Body_Of
private
  type Message is          -- no Subject component
    record
      Sender : Username;
      Receiver : Username;
      Msg_Body : Text;
    end record;
end Messages;
```

Or, if a message with a date and time stamp were desired, the abstraction could be augmented with an additional component, such as with the standard type Calendar.Time:

```
with Calendar;
package Messages is
  type Username is ...
  type Line is ...
  type Text is ...
  type Message is private;
  -- procedures Set_Sender, Set_Receiver, Set_Body,
  --         Set_Subject, and Set_Time
  -- functions Sender_Of, Receiver_Of, Body_Of,
  --         Subject_Of, Time_Of
private
  type Message is
    record
      Sender : Username;
      Receiver : Username;
      Time_Stamp : Calendar.Time;    -- new
      Subject : Line;
      Msg_Body : Text;
    end record;
end Messages;
```

Although the generic feature in Ada is not powerful enough to allow these variations as tailoring of a single common package, all of the Message package examples (as well as their corresponding bodies) could have been generated automatically, given the desired set of components for objects of type Message. This, therefore, becomes an example of generated reuse, where the generator is the reusable software and not the actual message package software. For example, a simple editor-substitution generator has been constructed which accepts input such as

the following and emits Ada equivalent to the example shown above.

```
Generate_Package
      (Context => "",
       Local_Decls =>
         "subtype username is string(1..10);" &
         "subtype line is string(1..60);" &
         "subtype text is string(1..80);",
       Package_Name => "messages",
       Private_Type => "message",
       Set_1 => "set_sender",
       Set_2 => "set_receiver",
       Set_3 => "set_subject",
       Set_4 => "set_body",
       Get_1 => "sender_of",
       Get_2 => "receiver_of",
       Get_3 => "subject_of",
       Get_4 => "body_of",
       Local_Type_1 => "username",
       Local_Type_2 => "username",
       Local_Type_3 => "line",
       Local_Type_4 => "text");
```

The effort to construct this generalization amounted to the writing of about 20 lines of software and the building of templates from the original unit. The effort to reuse the component is the construction of the above call. This could be seen as effort equivalent to declaring 17 string constants.

Note that, at this level of generality, which came at considerably higher cost than the previous tailoring, more than just a message package for a mail system could be generated. Any private type implemented as a record of components with set procedures and access functions could be generated with such a program. Therefore, this represents a domain-independent form of the component, where any mail system details are supplied by the reuser. So, the benefit of applying this substantial generalization effort is that the component can now be used by many domains. In fact, we will see that this same generator can be used to replace part of the Mailbox package, as well.

Although the style of the Mailbox package is not as general as the Messages package, there are several opportunities to make it more general and therefore more reusable in other contexts. For example, it could be tailored by making the constituent type Message and the maximum mailbox size generic formal parameters:

```
generic
   type Message is private;
   Max_Mailbox_Size : Natural := 1000;
package General_Mailboxes is
   ... -- same as package Mailboxes, above
end General_Mailboxes;
```

This arrangement of the Mailboxes package is shown in the bottom part of Figure 4b. Fortunately, no operations on the type Message were needed by the package Mailboxes, otherwise those operations would have had to have been passed as generic parameters.* Therefore, following the convention suggested above, the generalization effort here is the effort to write two generic formal parameter declarations. Reuser effort is the choice of a type and a value to perform the instantiation.

Beyond the relatively simple generalization shown above, it can be observed that the Mailbox abstraction is actually composed of a four-component record-type abstraction and an array. Reusing the previously described example of private record type abstractions, the package Mailboxes could be divided into two separate abstractions as follows:

```
generic
   type Message is private;
package General_Mail_Items is
   type Mail_Item is private;
   procedure Set_Message
      (An_Item : in out Mail_Item; To : Message);
   procedure Set_Read
      (An_Item : in out Mail_Item; To : Boolean);
   procedure Set_Answered ...
   procedure Set_Deleted  ...
   function Get_Message (An_Item : Mail_Item) return Message;
   function Is_Read (An_Item : Mail_Item) return Boolean;
   function Is_Answered (An_Item : Mail_Item) return Boolean;
   function Is_Deleted (An_Item : Mail_Item) return Boolean;
private
   type Mail_Item is     -- a modified implementation
      record
         Item : Message;
         Read : Boolean;
         Answered : Boolean;
         Deleted : Boolean;
      end record;
end General_Mail_Items;

generic
   type Mail_Item is private;
   Max_Mailbox_Size : Natural := 1000;
package General_Mailboxes is
   subtype Box_Size is Natural range 0 .. Max_Mailbox_Size;
   type Item_Array is array (Positive range <>) of Mail_Item;
   type Mailbox (Size : Box_Size := 0) is
      record
         Items : Item_Array (1 .. Size);
      end record;
```

*If Ada supported full inheritance, it would be possible to write the Mailbox abstraction so that it relies on certain operations to be defined for the generic formal type Message. The user would then guarantee that any expected functions would be available for any actual type parameter associated with the formal type Message, eliminating the syntactic complexity of passing them via additional generic formal subprograms. This illustrates one of the advantages of late binding, something that Ada disallows in order to ensure that required operations are available prior to the compilation of any instantiations of the generic.

```
procedure Store (Box : Mailbox; Owner : String);
  procedure Retrieve (Box : in out Mailbox; Owner : String);
  function Size (Of_Box : Mailbox) return Box_Size;
  procedure Remove
    (Mail_Item_At : Positive; In_Box : in out Mailbox);
  procedure Append
    (A_Mail_Msg : Mail_Item; To_Box : in out Mailbox);
  No_Msg_At_Position : exception;
end General_Mailboxes;
```

These packages are depicted by Figures 5b and 5c. In the above case, the client could obtain the functional equivalent to the original mailbox package via the following instantiations:

```
package Mail_Items is
    new General_Mail_Items (Messages.Message);
package Mailboxes is
    new General_Mailboxes (Mail_Items.Mail_Item);
```

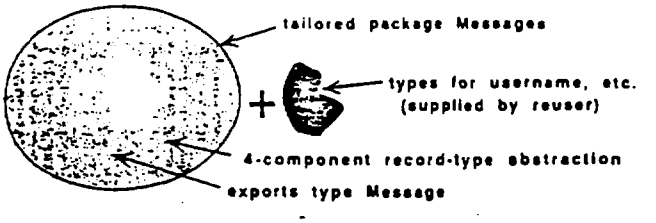


Figure 5a.

No additional changes are made during the second pass at tailoring the two packages. Only by generating the Messages package can the decisions about the structure of the abstract data type be generalized, since such a run-time tailoring is not possible within the Ada language.

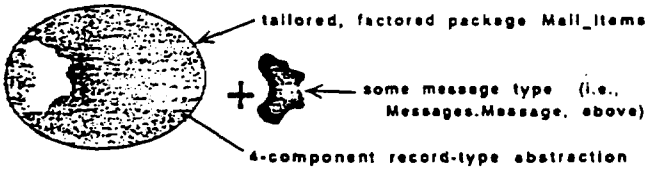


Figure 5b.

The Mailboxes package is broken into two components, one which implements Mail_Items as a record-type data abstraction, above.

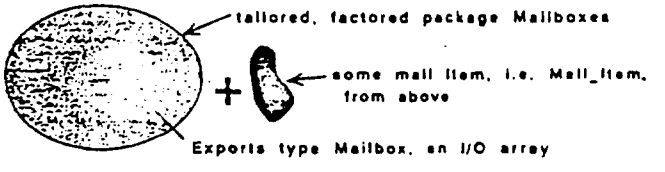


Figure 5c.

The other package factored from the original Mailboxes package implements an I/O list of mail items. This no longer contains any problem- specific function other than implement lists, so it can be replaced with a general-purpose list abstraction, as shown in the text.

Two tradeoffs in this example are observed. First, the specific way in which package Mail_Item was structured originally was modified into the more general multi-component record shown here. This tradeoff was accepted in order to allow this implementation of Mail_Items to be similar to the implementation of Messages, which was previously shown to be highly generalizable. This is an example of how standardization limits the choices available to the implementer while increasing the generality of the resulting programs. For example, by adopting this approach, the generator program mentioned before could be used to generate an equivalent package to Mail_Items through the following input, thereby allowing the generation of both the Messages package and the Mail_Items package from the same reusable component:

```
Generate_Package
        (Context => "with messages;",
         Local_Decls =>
          "type message is new messages.message;",
         Package_Name => "mail_items",
         Private_Type => "mail_item",
         Set_1 => "set_message",
         Set_2 => "set_read",
         Set_3 => "set_answered",
         Set_4 => "set_deleted",
         Get_1 => "get_message",
         Get_2 => "is_read",
         Get_3 => "is_answered",
         Get_4 => "is_deleted",
         Local_Type_1 => "message",
         Local_Type_2 => "boolean",
         Local_Type_3 => "boolean",
         Local_Type_4 => "boolean");
```

The second tradeoff was to make the type Mailbox visible. This was necessary since the client software will have to gain direct access to a Mail_Item within a mailbox array in order to perform the operations from package Mail_Items on it. Simply returning a value of Mail_Item via a function call would not allow the user to set the components of a Mail_Item in a mailbox. An alternative solution would have been to implement the items in a mailbox as access values, each designating a Mail_Item. In this way, a function returning an access value would provide the capability for the client to modify the designated object, a Mail_Item. This situations occurs frequently when factoring composite abstractions into their constituent abstractions, and suggests that by presenting objects directly on the interface to an abstraction, rather than just their values, an abstraction can be made more general and reusable.

Further generalizations are not shown in detail in the interests of space. However, note that the above General_Mailboxes abstraction is the only remaining custom-made application code in the example. It amounts

to an ordered list of items of discernible size, to which items can be appended and from which items can be deleted, and which can be stored to and retrieved from files. Except for the ability to store and retrieve the lists, such an abstraction would probably be available in a library of generic data structures. Assuming the constituent objects are private and not limited private, it would be possible to perform binary input/output on them. So, it is not unreasonable to augment an existing generic abstraction to include storage and retrieval. Such an augmentation of a list resource could be accomplished by layering something like the following onto it.

```
-- Layering on a list abstraction:
with Simple_Lists;
generic
   type Item is private;
   type Item_Access is access Item;
package General_Mailboxes is
   package Item_Lists is new Simple_Lists (Item, Item_Access);
   type Mailbox is new Item_Lists.List;
   procedure Store (A_Box : Mailbox; To_File : String);
   procedure Retrieve
      (A_Box : in out Mailbox; From_File : String);
end General_Mailboxes;
```

To obtain the equivalent functionality as was provided by instances of the earlier package General_Mailboxes, the following declarations would now be required:

```
package Mail_Items is
        new General_Mail_Items (Messages.Message); -- same
   type Mail_Item_Access is access Mail_Items.Mail_Item;
   package Mailboxes is new General_Mailboxes
                (Item => Mail_Items.Mail_Item,
                 Item_Access => Mail_Item_Access);
```

The client can treat the above package Mailboxes similarly to the earlier version; it will have all the same operations due to the derivability of those already implemented by Simple_Lists. Also, note that the mailbox implementation has been made private again by using designated objects to hold mail items. This would allow an Item_At function to return an access value to the actual mail_item and not just the value of that mail_item. This allows updates of the item via the operations that were defined in the Mail_Item package (Set_Message, Set_Deleted, etc.).

Measurement Summary

Measurement is required at two points of the software cycle. When candidate units are being identified and domain-specific details are being distinguished from problem-specific details, estimates of the generalization effort necessary to remove any give detail are required. At the time of reuse, estimates of the configuration effort necessary to adapt a component for reuse are required. Observations from conducting several generalizations have shown that an initial estimate based on an ordinal scale is possible. This scale has value parameterization as the easiest to perform for both generalization and reuse. Harder than this is type or operation parameterization, which requires tailored generalization in the case of Ada. The hardest form of generalization is building a special-purpose component generator. This can be made easier through the use of code-generation support tools.

After an initial evaluation of the generalization effort has been made and an approach to generalization has been determined, a more accurate assessment of the effort may be possible. The most direct indicator of the effort required is the number of lines of code that have to be written, changed or added. In many cases, a generalization can be accomplished with just a few lines of new or changed code. However, in the case of unsupported component generation, the entire generator may have to be written.

Reuse effort is easier to quantify since the component in question is already known. The effort to configure a generator or to instantiate a generic can be estimated based on the number of inputs or parameters required. In most cases, the usage of a tailored or generated component is similar regardless of whether the component was developed from scratch or obtained from a repository. However, even this step can be complicated by the fact that a development might choose to be constrained in some way in order to take advantage of an available component. The costs of such a decision can be especially difficult to estimate. In the long run, however, it is expected that the adoption of a component, similar to the adoption of a standard, is a cost-effective choice.

Another measure that is needed is an estimate of the future value of a unit in a repository. It may not be the best approach to populate a repository with many units which were inexpensive to generalize if they will rarely be needed. It would be better to spend the time performing a difficult generalization if the resulting unit will more than return that investment. Here again, domain experts will have to assist in making this determination.

Future Work

Progress is needed on metrics to quantify generalization and reuse effort. Effective metrics will open the way to establishing an economic model of reuse that could enable an organization to choose its optimal approach to reuse engineering. Note that the same approach or even the same specific model would not necessarily be best for two different organizations. One obvious reason for this is that one organization may concentrate in a single application domain while another organization may do work in many

domains with very little repetition. The first organization may find its optimal approach to reuse is to develop a mature repository of domain-specific components while the second organization may find that only domain-independent components are likely to be cost effective.

In addition to the costs of generalization and reuse, an economic view of the software cycle suggested in this paper would have to deal with repository maintenance, component retrieval, component probabilities of reuse and cost savings, and the effort required of domain experts and repository experts. Current progress is being made in some of these areas by interviewing experts at one branch of the NASA Goddard Space Flight Center where reuse has been practiced for many years, originally with Fortran and more recently with Ada. The results of these interviews will assist us in formulating a more quantifiable model of the costs and benefits of reuse at that organization. It is hoped that this experience can then be extrapolated into a broader model of reuse engineering that can be adapted for use at other organizations.

References

[Bailey and Basili] J. Bailey and V. Basili, "Software reclamation: Improving Post-Development Reusability," in Proceedings Eighth Annual Conference on Ada Technology, Atlanta, Ga., 1990.

[Basili and Caldiera] V.R. Basili and G. Caldiera, "A Reference Architecture for the Component Factory," *Computer Science Technical Report Series*, University of Maryland, College Park, MD, March 1991, UMIACS-TR-91-24 or CS-TR-2607.

[Basili and Rombach] V.R. Basili and H.D. Rombach, "Support for Comprehensive Reuse," Software Engineering Journal, July 1991, (also, *Computer Science Technical Report Series*, University of Maryland, College Park, MD, February 1991, CS-TR-2606 or UMIACS-TR-91-23).

[Caldiera and Basili] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, Vol.24, No.2, Feb.1991, pp.61-70.

[Ellis and Stroustrup] M. Ellis and B. Stroustrup, "The Annotated C++ Reference Manual," Addison Wesley, 1990, p. 341.

[AIC] Ada Information Clearinghouse. "STANFINS-R - COBOL and C Programmers Moving Successfully to Ada." *Ada Information Clearinghouse Newsletter 8*, 2, June 1990.

John W. Bailey is a Ph.D. candidate at the University of Maryland Computer Science Department. He has been consulting and teaching in the areas of Ada and software measurement for nine years, and is currently consulting to Rational. He has an M.S. in computer science from the University of Maryland, where he also earned a bachelor's and a master's degree in cello performance. He is a member of the ACM.

Victor R. Basili is a professor at the University of Maryland's Institute for Advanced Computer Studies and Computer Science Department. His research interests include measuring and evaluating software development and is a founder and principal of the Software Engineering Laboratory, a joint venture of NASA, the University of Maryland, and Computer Sciences Corporation. He received a B.S. in mathematics from Fordham College, an M.S. in mathematics from Syracuse University and a Ph.D. in computer science from the University of Texas. He is a fellow of the IEEE Computer Society.

On the Nature of Bias and Defects in the Software Specification Process

PABLO A. STRAUB
COMPUTER SCIENCE DEPARTMENT
CATHOLIC UNIVERSITY OF CHILE

MARVIN V. ZELKOWITZ
DEPARTMENT OF COMPUTER SCIENCE AND
INSTITUTE FOR ADVANCED COMPUTER STUDIES
UNIVERSITY OF MARYLAND AT COLLEGE PARK

Abstract

Implementation bias in a specification is an arbitrary constraint in the solution space. This paper describes the problem of bias and then presents a model of the specification and design processes describing individual subprocesses in terms of precision/detail diagrams, and a model of bias in multi-attribute software specifications. While studying how bias is introduced into a specification we realized that software defects and bias are dual problems of a single phenomenon. This has been used to explain the large proportion of faults found during the coding phase at the Software Engineering Laboratory at NASA Goddard Space Flight Center.

1 Introduction

Most informal software specifications are ambiguous, imprecise, and incomplete. Moreover, this is usually not evident by looking at a particular specification. This has prompted research on desirable and undesirable characteristics of specifications and specification languages. To make specifications precise, formal languages are used. Some of these languages are defined so that automatic compilation or execution is possible. However, much detail has to be included in executable specifications [5]. This extra detail not only makes the specification harder to read [6], but also leads to 'implementation bias'.

Alas, implementation bias—an arbitrary constraint in the solution space—is a term often used but not well defined. This has resulted in two effects: Either (1) specifications are biased, or (2) they are incomplete, for fear of bias. In fact, what has been called 'bias' in the literature is sometimes the desirable record of design constraints and design decisions. The problem of bias is related to the more important problem of software defects, because both are manifestations of either misconceptions with respect to the problem or preconceptions with respect to the solution; hence, we study these two problems together.

OVERVIEW OF THE PAPER. This paper presents a model to help understand bias in software specifications. The remaining of this introduction presents our framework, the problem of bias and the concept of specification correctness. The next section presents our view of the process of specification and design. Section 3 presents our model of bias which is based both on the specification process and on a classification of requirements. Within this model, bias is not an absolute property of a specification, but depends on the process of creation of the specified requirements, that is bias depends on the process of specification and design. Section 4 presents the relationships that exist between bias and defects in a specification, and a study made at the Software Engineering Laboratory that explains the high relative incidence of coding faults in that environment.

1.1 Specification Framework

In this work we are considering multi-attribute specifications developed by starting from a description of requirements, and then refining it in several stages [3, Chapter 1]. Each stage takes a specification and produces a product, which is a more refined specification, until a program (i.e., a specification for a computation) is obtained. This view is not an endorsement of any particular development method: it models top down development, the waterfall life-cycle model, Boehm's spiral model, transformational programming, and other development methods.

We first define some related concepts.

Attribute: feature or dimension that characterizes software systems (e.g., average response time).

Requirement: constraint in the values of attributes (e.g., average response time shall be 0.5 seconds).

Preference measure: a measure of the goodness of the different values for a given attribute (e.g., smaller response time values are better).

Specification: statement of attributes, requirements, and preference measures for a software system.

Specificand set: set of all systems that satisfy a specification.

Solution set: set of all systems that solve a problem.

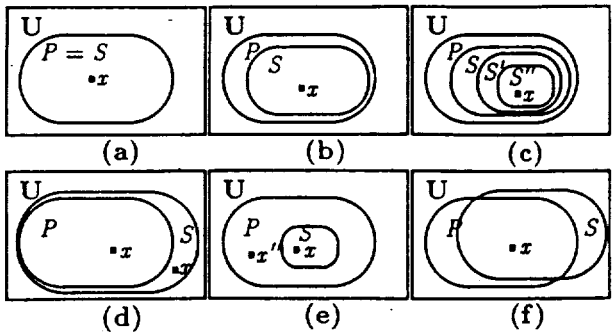


Figure 1: Specificand S, solution P, and particular solutions x and x'': (a) ideal, (b) acceptable initial specification, (c) successive specification stages, (d) incomplete specification, (e) bias, (f) usual case.

Whereas the specificand set is defined in terms of a particular precise specification of a problem, the solution set is defined in terms of the problem itself without reference to any written specification. That is, the specificand set comprises all systems that are correct with respect to the written specification, and the solution set comprises all systems that satisfy the user or customer. The differences between these sets are at the heart of our model; they are also the cause of defects in specifications.

1.2 The problem of bias

An ideal initial specification is general and precise enough so that a software system satisfies the specification if and only if it solves the problem at hand, that is, the specificand set equals the solution set (Figure 1a). This view is too optimistic, because there can be many solutions that do not even involve software. In practice, we only require software systems satisfying the specification to be solutions, and that no substantial class of solutions does not satisfy the specification, so that we can arrive at an optimal or nearly optimal solution (Figure 1b). An idealized development by staged specifications constrain the specificand set (Figure 1c) by adding design decisions—and nothing else. Incomplete specifications (Figure 1d) may lead to defects; for instance, x' satisfies the specification but it does not solve the problem. On the other hand, bias (Figure 1e) may lead to inefficiencies (e.g., optimal solution is really x'') and other development problems because the developers are overconstrained. Unfortunately, most specifications suffer both problems (Figure 1f).

A specification is biased if some of its requirements are arbitrary. Biased specifications overly constrain the specificand set, precluding some valid implementations as solutions to the problem at hand. Hence, the amount of bias is a common yardstick to judge software specification methods: those that are considered biased are usually rejected. Unfortunately, bias is sometimes confused with intended constraints in the solution set.

1.3 Avoiding bias

A generally accepted rule to avoid bias is "A specification should describe only *what* is required of the system and not *how* it is achieved."[1] However, this rule does not solve the problem: it only shifts it, because whether some requirement is a *what* or a *how* depends on one's point of view. For instance, the same requirement can be seen as a *how* by the designer and as a *what* by the implementor. During the process of refining the specification, some *how*'s become *what*'s: a design decision (i.e., *how* to do something) made by a designer is a requirement (i.e., *what* to do) for the implementor. A *how* becomes a *what* when a decision is made: a new requirement is incorporated into the current specification stage.

Consider a specification for a subprogram. The external interface of the subprogram is considered a requirement by the programmer (it is a *what*), because he or she cannot change it. This same interface was previously a *how* for the designer of the whole program, because he or she could have chosen an alternative interface. On the other hand, internals of the subprogram (e.g., algorithms, data structures, local variable names) are mostly *how*'s for the programmer, because he or she can change them.

There is no reason to include a *how* in a specification: specifications should describe what is desired and no more. However, often some attribute that is already fixed (i.e., it is a *what*) is not specified because of fear of bias. For instance, if within an institution there is a convention for local variable names for the purpose of easing maintenance, then the adherence to this convention is a *what*: It is already fixed, the programmer cannot change it, so it should be specified. We argue that this kind of constraint is not bias; in Section 3.3 we provide a definition of bias that is consistent with this view.

1.4 Specification Correctness

Specification bias and specification defects are intimately related. As can be seen from Figure 1, bias is related to the set difference of the solution set and the specificand set, $P - S$. That is, there is bias only if there are acceptable and preferred solutions outside the specificand set. Conversely, defects are related to the specificand set minus the solution set, $S - P$. That is, if an implementation i is unacceptable but is correct with respect to the specification, it is in the set difference (i.e., $i \notin P \wedge i \in S \Rightarrow i \in S - P$). In other words, bias and

[1] A common statement of this rule is "A specification should describe only *what* the system should do, not *how* it should do it." This modified rule is only useful with functional specifications: it views a software system as a specification for a computation, rather than as a product.

defects in the specification are dual problems.

Assume that for a given specification, the specificand set is contained in the solution set. In this case, all correct implementations are acceptable. This motivates the notion of specification correctness with respect to a problem, which is similar to the more familiar notion of implementation correctness with respect to a specification. (The main difference between these two concepts is that specification correctness cannot be formally verified because it is defined relative to an abstract problem.) A specification is correct if it is realizable (there is a correct implementation) and complete (all correct implementations solve the problem). That is, for a correct specification it is possible to derive an implementation and any implementation derived solves the problem. On the other hand, a specification is called impertinent to the problem if there is not a correct implementation that solves the problem.

The above is formalized as follows: Let S be the specificand set of a specification and let P be the solution set of a problem.

- The specification is *realizable* iff $S \neq \emptyset$.
- The specification is *complete w.r.t. the problem* iff $S \subseteq P$.
- The specification is *correct w.r.t. the problem* iff it is realizable and complete.
- The specification is *pertinent to the problem* iff $S \cap P \neq \emptyset$.

The following relations between these concepts are immediate: correctness implies pertinence ($S \neq \emptyset \wedge S \subseteq P \Rightarrow S \cap P \neq \emptyset$); pertinence implies realizability ($S \cap P \neq \emptyset \Rightarrow S \neq \emptyset$); completeness and pertinence imply correctness (because pertinence implies realizability); unrealizability implies completeness and impertinence ($S = \emptyset \Rightarrow S \subseteq P \wedge S \cap P = \emptyset$); there is no correct specification for a problem without a solution ($P = \emptyset \Rightarrow \nexists S: S \neq \emptyset \wedge S \subseteq P$).

To analyze the correctness of a specification with respect to a problem, consider the emptyness of the set $S-P$, related to the completeness of the specification, and of the set $S \cap P$, related to the pertinence of the specification. There are four cases: (a) The specification is unrealizable; (b) the specification is correct; (c) the specification is realizable but not pertinent; and (d) the specification is pertinent but incomplete, that is the specification can be made correct by adding more requirements. Figure 2 presents these cases, with case (d) comprising two subcases. In cases (a) and (c), the only choice is to backtrack, since at this point it is impossible to derive an acceptable solution. In case (b) there are no problems of correctness, but there can be problems of specification bias, if the preferred solution lies outside the specificand set as in Figure 1e. In case (d), the specification is incomplete, so addition of problem-specific information is needed to achieve a correct specification.

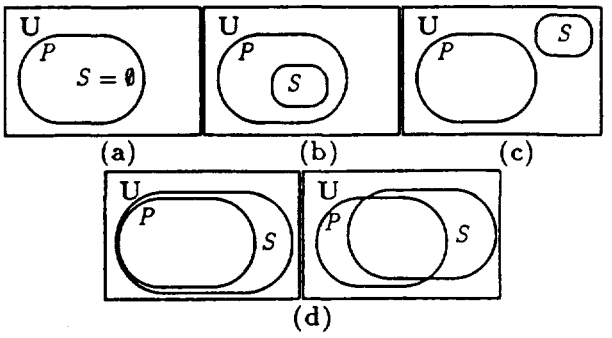


Figure 2: Specificand set S with respect to solution set P: (a) unrealizable, (b) correct, (c) realizable but impertinent, (d) pertinent but incomplete.

2 Specification Refinement

The specification and design processes are complex processes in which technical knowledge, art and inspiration take part [10]. Goel and Pirolli [4] describe the traditional view of design as a four-step process: "(1) an exploration and decomposition of the problem (that is, analysis); (2) an identification of the interconections; (3) the solution of the subproblems in isolation; and (4) the combination of the partial solutions taking into account the interconnections (that is, synthesis)."

In this work we go beyond these general processes and describe the subprocesses that occur specifically in software design. We characterize these subprocesses by how a current specification is updated to produce the next specification within a series, and also by how precision and detail are added to the specification. There is no assumption that all requirement analysis is done before design; on the contrary, requirements gathering and design are supposed to be intertwined [12].

2.1 Refinement Subprocesses

We assume that there is a written initial specification and that successive specifications will be created by a series of modifications to that specification. With respect to the subprocesses that perform these modifications—typically additions to the current specification—we postulate that there are four main kinds of activities that modify a specification:

Explication: addition of a requirement by making explicit a nonexplicit requirement.

Design decision: addition of a requirement by choosing a particular design.

Presentation change: change in the notation, presentation, or structure of the specification.

Retraction: withdrawal of a requirement from a previous decision or explication.

Even though we present these as discrete changes, actual changes to a specification usually involve a combination of them. For example, after finding an incorrect explication an analyst may replace the corresponding requirement by another one: a retraction followed by an explication.

Explication

Explication is one of the main activities during requirements gathering. Explications make the specification more complete, that is, ensure that software systems satisfying the specifications are solutions. In Figure 1 the goal is to transform a specification like (d) into one like (a). This goal is achieved by making explicit either *domain information*, *problem-specific information*, or *consequences of the specification*, thus reducing the specificand set.

Of course, the new requirement is not always a valid explication (e.g., something believed to be a consequence of the requirements might not be). This is intimately related to the concepts of specification correctness (Section 1.4) and bias (Section 3.3).

Design Decisions

As the name suggests, design decision is the most important process during design activities. Design decisions guide the implementation process towards a preferred set of solutions reducing the specificand set (as in Figure 1c). The information needed to make design decisions comes mainly from the previous specification and the solution domain. For example, semantic-preserving transformations in transformational programming are design decisions, because they preserve the functionality while improving other attributes of the algorithm.

We have identified several kinds of design decisions: decomposition, refinement, composition, abstraction, instantiation, reuse, creation of alternatives, and choice. Some of these are intimately related so we discuss them together.

Decomposition and refinement. Decomposition consists of dividing the problem into subproblems. It is usually followed by refinement, which means defining unspecified concepts or objects. These two processes are the core of stepwise refinement.

Composition. On the other hand, composition is the process of creating a solution to a problem by combining solutions to subproblems. That is, composition is the main process in bottom-up development. Composition is used most effectively in combination with reuse.

Abstraction, instantiation, and reuse. Abstraction as a design decision consists of specifying a solution to a more general problem (i.e., a problem of which the problem of interest is an instance), usually defining a set of (formal) parameters to describe particular instances. The rationale for solving more general problems is that it is often easier to abstract away particulars of the problem of interest and solve a general problem. Furthermore, the more general solution can be reused in other contexts.

Reuse as a design decision consists of prescribing the use of a particular solution to a subproblem. If the solution to be reused is parameterized (i.e., it has formal parameters) actual parameters must be provided to do the reuse. Instantiation is the process of defining actual parameters for a parameterized abstract solution.

A solution to reuse need not be already implemented: it may be simply specified as the solution to another subproblem. When several subproblems in the current design are instances of a single general problem, abstraction, instantiation and reuse can be employed to "factor" the design.

Creation of alternatives and choice. When it is not immediate which kind of design is the best, it is possible to create several alternative designs using some of these techniques. A valid implementation must conform to one of the created designs. After more elaboration of these designs, some are discarded until one design prevails. Choice is the process of selecting among alternative designs; the choice process is more objective when it is based on preference measures [2].

Presentation Changes

Presentation changes are intended to change the precision, formality, readability, modularity or other aspects of the specification itself, without affecting the specificand set, that is, without adding more information. For example, a condition written in English, referring to a collection of objects can be replaced by a logical predicate in which the collection is represented by a set.

Ideally, a presentation change does not change the specificand set, that is, it does not create new requirements. However, restrictions in the specification languages or methods used may impose additional constraints. In the above example, should our specification language support lists but not sets, we might have specified a list as an implementation for a set. If we later coded this list in Pascal we might have coded our list specification into an array or linked structure rather than the more efficient set data type that actually was originally specified. That is, as a result of a specification language deficiency we have added an additional arbitrary constraint for the program that resulted in it being less efficient, that is, we have added bias.

Retraction

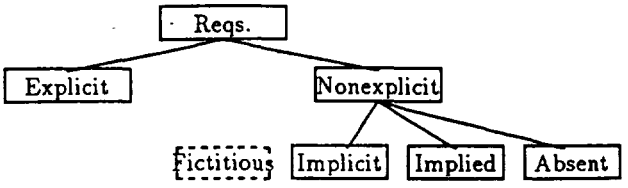


Figure 3: Classification of requirements: explicitness. Fictitious requirements are shown with segmented line because they are not real requirements.

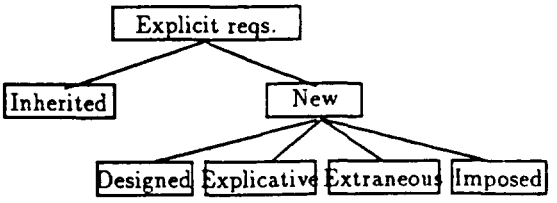


Figure 4: Classification of explicit requirements: origin.

Retraction occurs when a designer realizes that the current design is incorrect or otherwise undesirable. The goal of retractions is to create a *pertinent* specification, as defined in Section 1.4. As we said before, the retraction process is usually done in conjunction with other processes that create a new "replacement" requirement.

3 A Model of Bias

Presence of bias cannot be determined from the requirements alone, because it depends on the origins of requirements. For instance, if the origin of a particular requirement is in the problem, the requirement is not bias; if the origin is a misconception it may be. Hence, our definition of bias is based on a classification of requirements.

Requirements are classified into several classes with subtle differences. These subtleties are what makes bias hard to define and even harder to find. The main classification criteria we consider are explicitness and origin, which depends on the process of creation of new requirements.

3.1 Explicitness

A requirement is *explicit* if it is present in the specification; otherwise, it is *nonexplicit*.

Nonexplicit requirements are a recurring cause for misunderstandings in product development. They are further classified as follows (Figure 3).

Implicit requirements are those that are understood to be part of every product in the application domain, and so they are left unstated.

Implied requirements are logical consequences of other requirements.

Absent requirements are requirements unintentionally omitted in the specification, but are required by the solution set. These are not part of every product in the application domain.

Fictitious requirements [8] are assumptions made by the reader of the specification and not requirements at all: the reader believes that they are either implicit, implied or absent requirements.

A *real* nonexplicit requirement is either an implicit, implied, or absent requirement.

3.2 Origin

An explicit requirement is *new* with respect to a certain specification stage if it is first made explicit at that stage; otherwise, the requirement is *inherited* from previous stages. (When the specification stage is clear from context we will say simply 'new' or 'inherited' requirement.) Of course, every explicit requirement is new to one stage, namely the stage in which it is introduced.

The discussion in Section 2 motivates the following classification of new requirements with respect to their origin (Figure 4).

Designed requirements are the consequence of design decisions taken at the current specification stage.

Explicative requirements are created by explication of implicit, implied, or absent requirements.

Extraneous requirements are created by explication of fictitious requirements.

Imposed requirements are those imposed by the limitations of the specification method or language used, created as a side effect of a presentation change.

This classification describes possible origins for the requirements, but it does not provide a method to determine the origin. For example, without a complete analysis of the application domain, there is no definite method to tell whether a requirement is extraneous or the explication of an implicit requirement.

3.3 The Nature of Bias

We define bias in terms of the origin of the requirements described in a specification: A specification containing extraneous or imposed requirements is *biased*.

This definition provides insight into the problem of bias, including both its origins and consequences. The origin of bias is either wrongful interpretation of nonexplicit requirements or the limitations imposed by the specification method. The consequences are that the specificand set can be overly constrained or that the solution adopted can be suboptimal. That is, a biased specification will lead the design towards particular implementations that are not necessarily the best possible.

The definition does not provide a method to measure bias content in a specification, because bias is defined in terms of the origin of requirements and we cannot be completely sure of the origin of some requirements. Furthermore, bias is relative to the application domain and the software engineering environment, because the domain and environment define what is implicit.

For example, in an environment in which all programs are written in a particular programming language, the presence of idioms of this language in a specification is not necessarily bias, unless another implementation language is introduced to the environment. This is what happened at the Software Engineering Laboratory (SEL) at the National Aeronautics and Space Administration (NASA).[2] During the first experience with development in the Ada language they realized that software specifications for satellite dynamics simulators were "heavily biased toward FORTRAN. In fact the high level design for the simulators is actually in the specifications document" [1]. This was not a problem—on the contrary, it facilitated both development and reuse of specification and code—until the first development in Ada: the specifications had to be rewritten first. Given our definition of bias these FORTRAN-oriented specifications were not necessarily biased; they contained many designed requirements. Before Ada was introduced, the use of FORTRAN was an *implicit* requirement. After that, the choice of appropriate language became an *explicit* attribute, resulting in the assumption of FORTRAN as a *fictitious* requirement.

The relative nature of bias is an essential characteristic. It stems from the existence of nonexplicit requirements and the inherent uncertainty with respect to those requirements. That does not imply that there is nothing to do: an obvious task is to make explicit as much as possible about the domain and environment. If this is done, we are reducing considerably the possibilities of bias. However, as long as there are nonexplicit requirements, there will be doubt about these requirements and hence possibility of bias. Making explicit the implicit requirements of a certain domain and environment still leaves two sources of bias: restrictions on the method and languages, and absent requirements. These two cannot be avoided completely: the first because any method that provides some guidance in the specification process will guide the design to some particular kind of solutions; the second because at the beginning of a project most requirements are absent.

[2]The SEL was created in 1976 to study and improve the software process at NASA Goddard Space Flight Center.

4 Software Defects

Both bias and software defects are a consequence of problems in the development process. Section 1.4 shows the duality of bias and faults by analysing the differences of the specificand set and the solutions set. Here this comparison is extended further. We classify software defects in three classes [11]: *faults* occur in documents, *errors* occur in human processes, and *failures* occur in automatic processes. There is an analogy between the problem of bias and defects: fictitious requirements are like errors (both during human processes), imposed and extraneous requirements like minor faults (both occur in documents), and inefficiencies like minor failures (both occur during automatic processes). The criticality of the attributes involved is related to whether something is considered a fault or simply bias.

During software development, successive specifications are written, usually starting from an incomplete specification towards a correct specification. Every specification inherits from all previous specifications, so if there is a new requirement that contradicts an explicit previous requirement the new specification is inconsistent and hence unrealizable. The only solution is to retract either the new requirement or previous requirements. Similarly, if a new requirement contradicts a nonexplicit real requirement the specification is made impertinent to the problem (i.e., it solves another problem); again, the only solution is to retract. All too often a specification is unrealizable or impertinent but this is not evident to the developers so no retraction occurs and development continues. This is a secondary but important source of defects.

We have studied these problems at the SEL. The software analyzed are ground support systems for unmanned spacecraft. Most systems are about 100K source lines FORTRAN programs, but a sizable percentage are now in Ada. The SEL has a database describing systems and their development processes made in the last 15 years. The analysis that follows uses data from that database, but only considers relatively recent data (since January 1, 1986), because the software process has changed.

Table 1 summarizes counts of change reports classified by type of change (e.g., requirement changes, fault correction) in all SEL projects. From the table, 49.4% of the changes are due to faults, 12.3% correspond to planned enhancements and 10.6% are due to requirements changes.

Table 2 summarizes counts of the changes due to the 8074 faults of Table 1, classified by source of fault. From the table, 74.8% of faults are related to coding and 16.3%

Type of change	Count	%
Fault correction	8074	49.4
Environment change	533	3.3
Improvement of user services	1205	7.4
Planned enhancement	2018	12.3
Presentation changes	1464	9.0
Requirement changes	1730	10.6
Other	1327	8.1
Total	16351	100.1

Table 1: Changes by type in SEL projects since 1986.

Fault source	All faults	
	Count	%
Requirements	76	0.9
Functional specification	242	3.0
Design	996	12.3
Subtotal specifications	1314	16.3
Code	6043	74.8
Previous change	714	8.8
Other	3	0.0
Total	8074	99.9

Table 2: Fault source in SEL projects since 1986.

of the detected faults are directly related to incorrect specifications (our definition of 'specification' includes three SEL phases: requirements, functional specifications, and design). This simple analysis demonstrates that up to 16% of all problems can be related to implementation bias in the specifications.

However, because requirements documents and their changes originate outside the SEL and within some requirements generation group at NASA, these changes are not considered faults in the specifications. If we assume that the 1730 requirements changes in Table 1 were indeed fault corrections, the total number of faults would be $8074 + 1730 = 9804$, the total number of specification faults would be $1314 + 1730 = 3044$ and hence specification errors would account for up to 31.0% of all faults. This assumption is not as extreme as it looks, because predicted changes in the requirements, improvements and environment (hardware) changes are classified separately. In summary, considering all faults, between 1/6 and 1/3 of all faults at the SEL are related to specifications, and potentially are related to implementation bias.

Another source of faults related to specifications are faults of omission: when something is not specified it is not a problem of the code but of the specification. The fact that the problem shows up during coding or testing does not mean that the problem is coding. Table 3 shows counts of faults of omission, commission, omission/commission separated by fault source (the 'Total' column is not identical to the 'All faults, Count' column from Table 2 because

Source	Comm.	Om.	Both	None	Total
Reqs.	19	40	8	9	76
Specs.	102	78	40	20	240
Design	253	550	159	34	996
Code	2302	2334	921	482	6039
Prev. chg.	289	295	79	50	713
Total	2965	3297	1207	595	8064
Percent	36.8	40.9	15.0	7.4	100.0

Table 3: Omission and commission faults in SEL projects.

10 faults had invalid data). At the SEL 37% of all faults are faults of commission, 41% are faults of omission and 15% are faults of omission/commission. Thus, about one half of the faults are of omission and potentially can be attributed to incompleteness in the specifications.

In conclusion, even though coding appears to be by far the most important source of faults, a deeper analysis of the specification process reveals that many coding faults have roots in earlier stages. Implementation bias undoubtedly plays an important role in many of these 3000 faults that are related to changes due to specification issues.

5 Conclusion

Even though bias is widely recognized as an undesirable property of specifications, it has not been adequately studied. This has caused confusion with the related concept of design decision, so that the presence of designed requirements in specifications has been considered undesirable. This is in contrast with the use of specifications in other engineering disciplines, where a specification may include many designed requirements (e.g., materials, manufacturing methods).

In this paper we presented a model to describe the nature of bias and distinguish bias from designed requirements and other requirements in a specification. This model is based on a classification of all the requirements described in a specification and also those that are not described (i.e., nonexplicit); it explains the nature of bias, but since it uses nonexplicit requirements it does not lead to any definite method to detect bias. However, the model does explain both the relative and unavoidable nature of bias. Because bias depends on the specification process we had to model that process. This modeling shed light on the problem of software defects, a relationship that in turn helped us to potentially explain the high relative number of coding faults found at the SEL.

Although we have developed an explanatory model of the design process, quantification of these concepts is needed before we can develop practical procedures for applying them in large scale developments. Additional work

in this direction in continuing.

Acknowledgements

This research was supported in part by grant NSG-5123 from NASA Goddard Space Flight Center to the University of Maryland. Thanks to Sergio Cárdenas-García and Eduardo Ostertag for their helpful comments. P. Straub was partially supported by a scholarship from the Catholic University of Chile while he was at the University of Maryland.

References

[1] Carolyn E. Brophy, W.W. Agresti, and Victor R. Basili. Lessons learned in use of Ada-oriented design methods. In *Proceedings of the Joint Ada Conference*, March 1987.

[2] Sergio Cárdenas and Marvin V. Zelkowitz. Evaluation criteria for functional specifications. In *Proceedings 12th Int'l Conf. on Software Engineering*, pages 26–33, Nice, France, March 1990.

[3] Bernard Cohen, William T. Harwood, and Melvyn I. Jackson. *The Specification of Complex Systems.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] Vinod Goel and Peter Pirolli. Motivation the notion of generic design within information-processing theory: The design problem space. *AI Magazine*, 10(1), spring 1989.

[5] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[6] C.A.R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, pages 85–91, September 1987.

[7] Cliff B. Jones. Systematic program development. In N. Gehani and A.D. McGettrick, editors, *Software Specification Techniques.* Addison Wesley, Reading, Massachusetts, 1986.

[8] Edward V. Krick. *An Introduction to Engineering and Engineering Design.* John Wiley and Sons, New York, N.Y., second edition, 1969.

[9] Harlan D. Mills, Michael Dyer, and Richard C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–24, September 1987.

[10] Ellen Shoshkes. *The Design Process.* Whitney Library of Design, New York, 1989.

[11] Pablo A. Straub and Eduardo J. Ostertag. EDF: A formalism for describing and reusing software experience. In *International Symposium on Software Reliability Engineering*, pages 106–113, Austin, Texas, May 17–18 1991.

[12] William Swartout and Robert Balzer. On the inevitable intertwinning of specification and implementation. *Communications of the ACM*, 25(7):438–440, July 1982.

An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity

Jianhui Tian
Soft. Eng. Process Group
IBM Canada Laboratory
North York Ontario,
Canada

Adam Porter
Computer Science Dept.
University of Maryland
College Park, Maryland

Marvin V. Zelkowitz
Inst. for Advanced Computer Studies
and Computer Science Dept.
University of Maryland
College Park, Maryland

Abstract

Identification of high cost modules has been viewed as one mechanism to improve overall system reliability, since such modules tend to produce more than their share of problems. A decision tree model has been used to identify such modules. In this current paper, a previously developed axiomatic model of program complexity is merged with the previously developed decision tree process for an improvement in the ability to identify such modules. This improvement has been tested using data from the NASA Software Engineering Laboratory.

1 Introduction

Identification of high cost modules has been viewed as one mechanism to improve overall system reliability, since such modules tend to produce more than their share of problems. In order to identify such modules, Selby and Porter [2, 3] developed a decision procedure based upon decision trees. With their technique, which we call Classification Tree Analysis (CTA), they showed on a set of 16 large-scale programs containing over 4700 modules obtained from the NASA Software Engineering Laboratory, that they could identify which subset of the 74 measures obtained from each module would produce good estimators of high-cost modules.

Recently Tian and Zelkowitz [4] developed an axiomatic model of program complexity. Based upon this model, the 74 measures kept on each of the 4700 modules could be reduced to only 18 measures that represented valid complexity measures. Using these 18 measures, the decision tree process results in an improvement over the original Selby-Porter method.

In this paper we will first describe the original decision tree process, we then summarize the axiomatic complexity model, and then demonstrate that we can improve on the previous model in identifying high-cost modules.

2 Classification Tree Analysis

In a series of earlier studies by Selby and Porter, a technique called classification tree analysis (CTA) was used to identify high cost components. Of critical importance to CTA is the selection of measures (or attributes) to construct the classification tree.

We define a high cost component as one in the uppermost quartile (i.e., 25 percent) relative to past data. The rationale for this definition is the so called "80:20 rule", which states that about 80 percent of a software system's cost is associated with roughly 20 percent of the system.

A classification tree is essentially a decision tree that branches on the range of values according to a measure at an internal node repeatedly until a component can be identified as high or low cost, or until all measures are exhausted.

The classification tree method that was used, called the classification paradigm, consists of the following three integral parts:

- **Classification tree generation** is the central

activity of constructing classification trees and preparing them for analysis and feedback;

- **Data management and calibration** are the activities that retain and manipulate historical data and tailor classification tree parameters to the development environment; and

- **Analysis and feedback** is the part that leverages the information resulting from the tree generation by applying it in the development process. The central piece of the application of classification tree is to develop remedial plans and take corrective actions.

2.1 CTA Method

The goal is to predict high cost modules in the current project with *high cost* being interpreted as the highest quartile. The historical data (or training set), consisting of one project immediately preceding the current one, are grouped into quartiles according to a measure's value, with all measures being considered.

Starting from the root, a measure is selected to separate modules into four subsets associated with each arc. The number to the left of an arc is the lower (inclusive) bound and the number to the right is the upper (non-inclusive) bound for the subset according to the measured value. So we have four subsets (quartiles).

A set of modules associated with an arc is positively identified if more than a threshold (termination criterion) of modules are in the highest quartile of cost, and it is represented in the tree as a terminal node marked with a "+" sign. A set can be negatively identified similarly, and represented correspondingly by a "−" sign. If a set cannot be either positively or negatively identified, another measure is selected to further classify these modules into finer subsets. This process continues until either all modules are identified or all measures are exhausted without being able to make such a determination. In the latter case, the terminal node is marked with a "?" sign, representing that CTA can not make a prediction for modules in this set.

Notice that the generation of the classification tree depends solely on the training set and various parameters selected for the technique. The current project will only use the tree but not affect the structure of the tree.

	Modules				
	m_1	m_2	m_3	m_4	m_5
cyclomatic complexity	3	8	13	30	45
module+function call	8	40	7	3	12
operators	30	18	10	33	58
module calls	3	4	3	0	5
prediction	−	?	−	−	+
actual	−	−	+	−	+

Table 1: Predicting High Cost Modules

As an example, consider the sample (fictitious) test data of Table 1, and the classification tree in Figure 1. This test set includes 5 modules and 4 measures. In this case, the CTA method predicts 3 out of 4 modules correctly (it misses module m_3) and is unable to classify module m_2 through the classification tree. For example, module m_5 follows the right most branch from the root (cyclomatic complexity of m_5 is greater than 26) and again follows the right most branch from there (operator counts of m_5 is greater than 34). We can finally predict it to be of high cost because its module call counts falls between 4 and 10.

2.2 CTA Cost

There are two types of cost associated with the CTA technique: the cost of building classification trees and the cost of using them. The former is determined by the factors: 1) the CTA parameters, 2) the size of the available measure pool where measures are to be selected, and 3) the implementation efficiency of the CTA supporting tools. For the latter cost factor, the tree size is a good measure. Because the classification trees we are studying have fixed structure (there are 4 branches from every internal node), we can effectively capture the cost of using classification trees by counting the number of internal nodes for them.

2.3 CTA Performance

According to the match between CTA predictions and actual cost data for the modules in a test set, various performance measures can be defined:

Coverage: The percentage of modules (either positively or negatively) identified;

Accuracy: The percentage of correct matches between predictions and actual data;

Comsistency: The percentage of predicted high cost modules who are actually high cost. High consistency

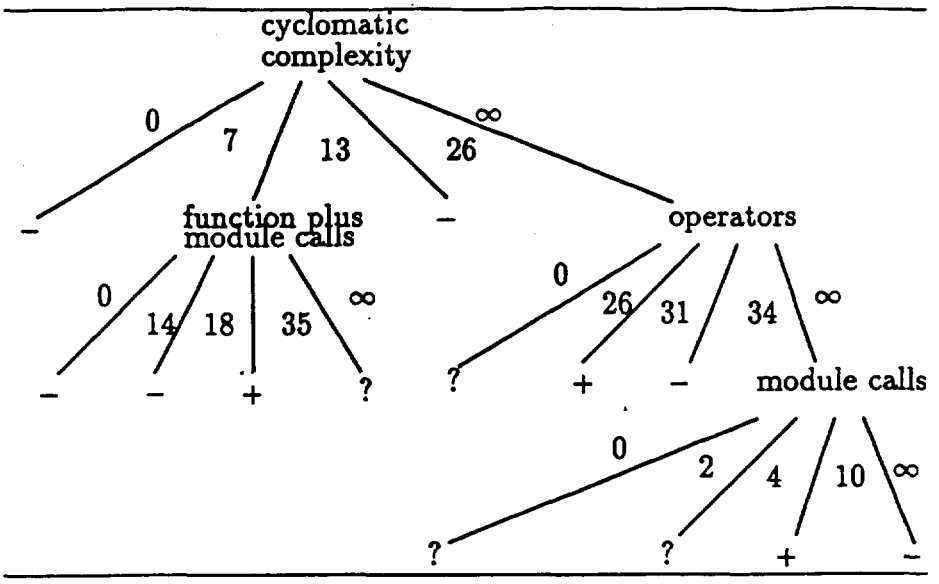


Figure 1: Component Classification Tree

indicates less "false alarms;" and

Completeness: The percentage of actual high cost modules predicted correctly by CTA. It reveals the power of CTA to uncover high cost modules.

3 Axiomatic Program Complexity

Most program complexity studies define complexity as a numeric comparison between any two programs. However, we have come to realize that some programs are inherently incomparable. For example, it makes litttle sense to compare the complexity between a payroll system and a real-time emission control system in a car. They each come from radically different application domains.

Instead we view *complexity* as a partial ranking among the set of programs and a *complexity measure* as a function applied to specific programs as an approximation of the attribute we are trying to determine. The following summarizes this model [4].

3.1 Axiomatic model

Consider a program as a hierarchy of modules consisting of instructions, data, and the underlying execution control mechanism. We initially limit ourselves to a Pascal-like nested scope sequential control language. Programs are represented by their abstract syntax trees:

- **U** represents the set of all programs.
- $AST(P)$ represents a binary abstract tree representation for program P. The root node of program P is given by $root(P)$, the left subtree of P is $left(P)$ and the right subtree of P is given by $right(P)$.
- For programs P and Q, $IN(P,Q)$ is true if P is a subprogram of Q (i.e., $AST(P)$ is a subtree of $AST(Q)$).
- If $IN(P,Q)$ is true, then $dist(P,Q)$ represents the path length in order to go from $root(P)$ to $root(Q)$.
- P with all free occurrence of x replaced by y not in P is denoted as P_y^x. We use P_β^α to mean the renaming is carried out for all corresponding one-to-one pairs in lists α and β, where

$$(var(P) - \alpha) \cap \beta = \emptyset$$

($var(P)$ is the variable list of program P).

A *complexity ranking* $\mathcal{R}$ is a binary relation on the set of programs. The complexity ranking between programs P and Q is $\mathcal{R}(P,Q)$. We interpret $\mathcal{R}(P,Q)$ as P being no more complex than Q. P and Q are *comparable*, denoted $C(P,Q)$, if either $\mathcal{R}(P,Q)$ or $\mathcal{R}(Q,P)$ holds, i.e., $C(P,Q)$ iff $\mathcal{R}(P,Q) \vee \mathcal{R}(Q,P)$.

A *complexity measure* $\mathcal{V}$ is a function that maps every program into a vector of real numbers: $\mathcal{V} : \mathrm{U} \rightarrow \boldsymbol{R}^n$.

Although simple definitions, we are immediately confronted by a difficult problem:

Theorem T1: There exist complexity rankings that are undecidable.[1]

Although the general problem of complexity ranking is undecidable, many practical rankings are not. In what follows we restrict ourselves to these more practical rankings.

Axiom A1: $(\forall P, Q)$ ($\boxed{\mathrm{P}} = \boxed{\mathrm{Q}} \Rightarrow \mathcal{C}(P, Q)$) where $\boxed{\mathrm{X}}$ is the function of program X.

Given programs P and Q, the problem of $\boxed{\mathrm{P}} = \boxed{\mathrm{Q}}$ is unfortunately also undecidable. This axiom, then, is at the center of the problem of developing effective complexity measures on real programs. We certainly want to be able to compare equivalent programs in order to determine which is best; however, undecidability says that we cannot always do this. It is for this reason that most complexity measures have not achieved significant breakthroughs since the underlying models are rarely comparable. However, in many practical applications, such as described above, we know or can assume that two given programs have the same or similar functionality.

Because of this, in practice we often use a weaker form of this axiom that only addresses the similarity of two programs:

Axiom A1': $(\forall P, Q)$ ($\boxed{\mathrm{P}} \approx \boxed{\mathrm{Q}} \Rightarrow \mathcal{C}(P, Q)$).

A program in general consists of many hierarchically related components. As a result, we require that a program must be comparable with a subpart of itself.

Axiom A2: $(\forall P, Q)$ $(IN(P, Q) \Rightarrow \mathcal{C}(P, Q))$

[1] Axiom and theorem references are keyed to [4], which also contains the proofs of the theorems. Some of the theorems given in that paper are not relevant to this present discussion and hence are not listed here.

Axiom A2 brings up the intuitive notion that we would like complexity to increase as programs become larger, i.e., if P is a component in Q ($IN(P, Q)$), then P is no more complex than Q. We left this out because there are cases where the opposite is true. Consider Q formed from P by addition of easily recognizable keywords or tags; Q might be more readable, thus easier to maintain as a result. Another case is that loops are often more easily understood if they include their initialization code than if presented without it.

Contextual information might help to reduce the complexity of composite programs. But the degree of the reduction must be limited, otherwise infinitely large programs paradoxically might be the simplest. On the other hand, a periodic function such as *cosine(x)* as the complexity of a program, where x is some size measure of a program P, is clearly not acceptable. As a general trend, then, adding components must result in a more complex program:

Axiom A3:
$(\exists K \in \boldsymbol{N})(\forall P, Q)((IN(P, Q) \wedge (dist(P, Q) > K)) \Rightarrow R(P, Q))$

Since our goal is to compare the complexity of two different programs, define a predicate $\mathcal{T}$ such that $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q))$ is true if program P is no more complex than program Q. For $\mathcal{V}$ into $\boldsymbol{R}$, we have the obvious definition that $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q))$ is just $(\mathcal{V}(P) \leq \mathcal{V}(Q))$. For higher dimensions, other results are possible (e.g., a dot product called the *performance level* measure which compares alternative software designs [1]).

$\mathcal{T}$ is our decision process which determines how well $\mathcal{V}$ approximates our complexity ranking $\mathcal{R}$ between P and Q based on the measured complexity values $\mathcal{V}(P)$ and $\mathcal{V}(Q)$. We would like the relationship to be $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q)) \Longleftrightarrow \mathcal{R}(P, Q)$, and in fact it is an implied axiom in most other complexity models. However, we believe that this is the major weakness that has prevented most complexity models from being truly effective. Because of undecidability issues (e.g. theorem T1), for all P and Q we cannot determine $\mathcal{T}$ for every $\mathcal{R}$. As a result, we use a weaker condition, namely:

Axiom A4: $(\forall P, Q)$ $(\mathcal{R}(P, Q) \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q))$

Since for many useful applications, $\mathcal{R}$ defines a total ranking, we then have:

Theorem T5: When $\mathcal{R}$ is total, i.e., $(\forall P,Q)\mathcal{C}(P,Q)$, we have:

$$(\forall P,Q)\ (\mathcal{V}(P) < \mathcal{V}(Q) \Rightarrow \mathcal{R}(P,Q))$$

In order to be useful, we would like our complexity measures to distribute programs across a range of values. If there is only a single "dominating" cluster point, we gain little information from the measure. Axiom **A5** allows, for rough comparisons, bi-polar or multi-polar distributions:

Axiom A5: $(\forall k \in R)(\exists \delta > 0)\ (|\mathrm{U} - \{P : \mathcal{V}(P) \in [k-\delta, k+\delta]\}| = |\mathrm{U}|)$

Axiom **A5** forces our complexity measure to be nontrivial, as in:

Theorem T7: $(\forall P)(\exists Q)\ (\mathcal{V}(P) \neq \mathcal{V}(Q))$

When $\mathcal{V}$ maps programs into a discrete bounded set S, axiom **A5** requires that at least two points in S have infinitely many programs with such values:

Theorem T8: If set S of complexity values is finite, then:

$$|\{k : (k \in \mathrm{S}) \wedge (|\{P : \mathcal{V}(P) = k\}| = |\mathrm{U}|)\}| \geq 2$$

3.2 A classification model

Given these five axioms, we developed a classification model for categorizing the various complexity measures depending upon the information they provide. A vertical classification uses a subset of the attributes for the entire program, while a hierarchical classification uses some functional relationship among the program's parts.

Vertical classification

A complexity ranking $\mathcal{R}$ is *abstract*, denoted $AB(\mathcal{R})$, if given P and Q with $AST(P) = AST(Q)$, then $\mathcal{R}(P,Q)$(and equivalently, $\mathcal{R}(Q,P)$).

If two programs are syntactically identical except for variable names, as long as two set of names are isomorphic, the only conceivable differences is interpretational (the meaning attached to each name). On the other hand, when considered functionally, each name is just a surrogate for the underlying data object. Thus we have the classification:

A complexity ranking $\mathcal{R}$ is *functional*, denoted $FN(\mathcal{R})$, if given P and Q with name sets α and β such that $AST(P^{\alpha}_{\beta}) = AST(Q)$, then $\mathcal{R}(P,Q)$.

Hierarchical classification

Assessing complexity by using only the components while ignoring interactions (i.e. ignoring the context where the components are defined and used) results in a *context free* ranking: A complexity ranking $\mathcal{R}$ is *context free*, denoted $CF(\mathcal{R})$, if given P, its ranking with respect to any given Q can be uniquely determined by: (1) Q and (2) $root(P)$, the complexity ranking of $left(P)$, and the complexity ranking of $right(P)$.

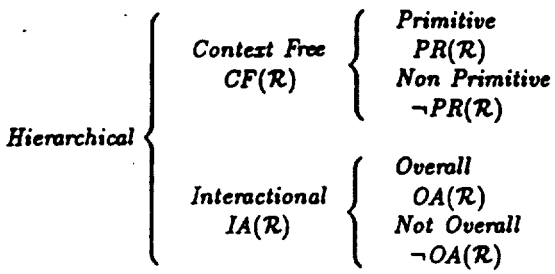
As a special case of context free complexity where organizational information is completely ignored, we can have primitive complexity: A complexity ranking $\mathcal{R}$ is *primitive*, denoted $PR(\mathcal{R})$, if all programs P and Q with the same collection of $AST(P)$ and $AST(Q)$ nodes (same number of occurrences for each corresponding pair) then $\mathcal{R}(P,Q)$.

Also, a complexity ranking $\mathcal{R}$ is *interactional*, denoted $IA(\mathcal{R})$, if it is not context free, i.e. $\neg CF(\mathcal{R})$.

Without considering interaction, the complexity of the composite complexity is the sum of all the components complexities. However, due to interaction among component parts, the total complexity may be greater than the sum. Such a complexity ranking is called *overall*.

If we are allowed to modify the internal structure, or reorganize the program according to some programming practices (such as modularization, data abstraction and information hiding), we may be able to cut down the interfacing complexity, thus the overall complexity. Since the two programs are functionally equivalent, they are comparable in complexity (**A2**).

The relationship among different hierarchical classes can be summarized in the following tree:



Using this model, we have been able to develop Weyuker's 9 properties for complexity measures as special cases of our axioms [5]. Since those properties have been widely studied over the past 4 years, and since we can model her properties with our classification model, we believe that our axioms are a reasonable approximation of program complexity.

4 Application of the Model

Sixteen software systems, ranging from 3000 to 112,000 lines of FORTRAN source code, were selected from NASA ground support software for unmanned spacecraft control developed in the NASA/GSFC Software Engineering Laboratory. Each required between 5 and 140 person-months to develop over a period of 5 to 25 months by 4 to 23 persons. Each project contains from 83 to 531 *modules*, totalling over 4700 modules. There are 74 attributes, each quantified by a specific measure, for each module divided into three broad categories: fault, effort, and style (or complexity).

For each application instance, one of the projects was used as a training project in order to develop the classification tree for the next project. This was repeated for the remainder of the 16 projects.

Five of the projects were of a greatly different size than the others (by more than a factor of 3). We deemed these to not fulfill **Axiom A1'** on similarly of functionality. This reduced the set of projects to 11 (and 10 data points) and are given as Group A in what follows. We used a different ordering of 6 of the projects in terms of training set to give us Group B (and 5 additional data points). CTA refers to the original Classification Tree Analysis process, while ACT refers to the Axiomatic Classification Tree process developed in this paper.

4.1 Measure Screening

From the set of 74 measures for each module, we first eliminate all measures that are not directly measureable from the modules themselves. Thus effort data, e.g., number of hours to develop the module, are eliminated. We also eliminated change and error data since they represent interactions among program components and the operational environment. We can therefore reduce the number of measures to 40.

All candidate measures satisfy axioms **Axiom A1'** (comparing functionally equivalent programs), **Axiom A2** (comparing component-composite pairs), **Axiom A4** (measures agree with their ranking), and **Axiom A5** (no single cluster). However, many of the measures do not satisfy **Axiom A3**, the general monotonicity axiom. These measures are averaging measure such as *assignment statements per 1000 executable statements*, which may be correlated with average effort per 1000 lines or so, but not with the total development effort. Therefore these measures will be eliminated. This reduces the candidate measures from 40 to 18, with the candidate measure set S being the left half of Table 2.

Both abstract and non-abstract aspects contribute to cost, so measures from any vertical class are potentially acceptable. On the other hand, as we are only considering cost and complexity at the module level, the hierarchical classification is not relavent. The analysis based on the measure classification scheme does not eliminate any measure for CTA in this case.

4.2 Aggregate Evaluation

Given 18 remaining measures that meet the boundary conditions based on the axioms and measure classifications, we next determine which of them best predicts total effort. The underline distribution, as we assumed, is a four region distribution (grouped into four quartiles) determined by historical data. A quartile of modules is positively identified if more than 75% of the modules (tolerance level: 25%) have the upper most quartile of effort. The negative sets can be similarly identified.

Let $m_i(\mathcal{V})$ $(i = 1, 2, 3, 4)$ be the number of modules in quartile i using measure $\mathcal{V}$; $p_i(\mathcal{V})$ be the proportion of modules in $m_i(\mathcal{V})$ belonging or to the upper most quartile of effort; and $n_i(\mathcal{V})$ be the rest proportion in $m_i(\mathcal{V})$ (therefore $p_i(\mathcal{V}) + n_i(\mathcal{V}) = 1$). As a result, a quartile is positively identified if $p_i(\mathcal{V}) \geq 0.75$, and

Meets Axiom A3	Fails Axiom A3
	assignment statements per 1000 executable statements
	input-output statement per comment
assignment statements	input-output parameters per comment
input-output statements	input-output statements per 1000 executable statements
input-output parameters	input-output statements per input-output parameter
source lines	input-output statements per 1000 source lines
comments	function calls per comment
source lines minus comments	function calls per input-output statement
executable statements	function calls per function plus module call
function calls	function calls per input-output parameter
module calls	function calls per module call
function plus module calls	module calls per comment
cyclomatic complexity	module calls per input-output parameter
operators	module calls per function plus module call
operands	module calls per input-output statement
total operators	function plus module calls per 1000 source lines
total operands	function plus module calls per input-output statement
decisions statements	function plus module calls per input-output parameter
format statements	function plus module calls per 1000 executable statements
origin	function plus module calls per comment
	cyclomatic complexity per 1000 source lines
	cyclomatic complexity per 1000 executable statements

Table 2: Attributes passing initial screening

negatively identified if $n_i(\mathcal{V}) \geq 0.75$.

To formulate the objective function for the aggregated selection, we need to evaluate the contribution of each quartile. We can weight them by the number of modules falling into the quartile. Therefore, we formulate our selection criteria as:

$$\max_{\mathcal{V},\mathcal{V}\in S} \left\{ \sum_{i}^{4} \{m_i(\mathcal{V}) * p_i(\mathcal{V}) + m_i(\mathcal{V}) * n_i(\mathcal{V})\} \right\} \quad (1)$$

for i ranging from 1 to $p_i(\mathcal{V}) \geq 0.75 \vee n_i(\mathcal{V}) \geq 0.75$

This selection criterion maximizes the number of modules in positively or negatively identified quartiles. For each of the quartiles neither positively nor negatively identified, another measure is selected using the same criterion. The process continues until all modules are identified or all measures are exhausted.

5 Results

We applied both the original CTA process and the modified ACT process to the 16 NASA projects broken down into the 11 projects of groups A and six projects of B. The following sections describe the results of this analysis.

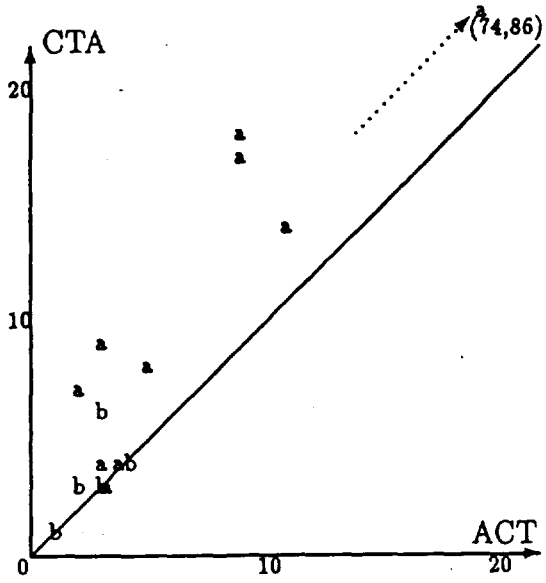
Size of generated trees

One measure of the efficiency of the technique is the size of the classification trees that are generated. Figure 2 shows that the axiomatic model (ACT) reduces tree size approximately 27% over the original CTA model from 188 nodes to 136 nodes in the 15 programs with average tree size dropping from 12.5 to 9.1 nodes.

The smaller the tree the more desirable (less costly to use to navigate through the tree, fewer measures to collect), thus a point in the upper left region represents an improvement over the original CTA.

Performance coverage

Table 3 compares the coverage based on the original and modified classification trees. In all the projects except one, near 100% coverage is achieved by both methods. Thus the decision tree analysis method almost always will predict a cost for a module and will



	individual data points			average		
	group A			group B	A	B all
CTA	17 15 7 8 86 4 9 3 4 18			3 6 3 4 1	17.7	3.4 12.5
ACT	9 11 2 5 74 4 3 3 3 9			3 3 2 4 1	12.3	2.6 9.1

Figure 2: Internal Node Count Comparison

	group A										group B				
CTA	98	98	99	98	91	93	97	100	100	98	100	98	97	97	100
ACT	99	100	97	100	82	93	100	98	100	99	98	98	100	97	100

a. individual data points

	group A	group B	all
CTA	97	99	97
ACT	97	99	97

b. average comparison

Table 3: Coverage Comparison

rarely leave modules unclassified. So, we can conclude that the CTA technique using either selection method achieves fairly good and consistent coverage, with an average of 97% coverage for both.

Performance accuracy

Accuracy improved about 5% with the ACT process, as given in Table 4.

Performance consistency

Table 5 gives the consistency comparison. This is the measure that drives the whole process, being that identification of high cost modules is the major goal

	individual data points			average		
	group A			group B	A	B all
CTA	66 76 78 63 53 67 71 85 73 71			70 50 81 77 58	70	68 69
ACT	67 73 80 66 50 67 81 83 73 89			79 54 86 85 58	75	74 74

Table 4: Accuracy Comparison

	individual data points			average		
	group A			group B	A	B all
CTA	70 66 31 54 52 63 30 16 50 10			7 100 33 17 65	39	35 38
ACT	67 61 37 57 56 63 50 15 50 23			43 85 40 29 65	50	50 50

Table 5: Consistency Comparison

	individual data points			average		
	group A			group B	A	B all
CTA	26 60 54 62 42 21 42 33 6 47			7 4 40 63 47	38	28 35
ACT	30 46 73 59 49 21 14 33 6 30			71 13 40 63 47	35	39 35

Table 6: Completeness Comparison

of the prediction process.

The performance level between the two selection methods is significantly different, with the modified ACT selection method outperforming the original CTA method by a margin of 50% to 38%.

Performance completeness

While ACT generates many fewer "false alarms," (i.e., predicting high cost modules which really are not high cost – the above consistency measure), both methods are comparable in actually identifying the high cost modules, i.e., the completeness measure of Table 6. That is, both will fail to indicate high cost modules in over half the cases.

6 Conclusions

Classification Trees are a method to use measureable quantities from program modules in order to determine desireable attributes from the development process. Identification of high cost modules should correlate closely with other process measures such as reliability.

In this paper, we presented a Classification Tree Analysis (CTA) method and a modification to it, the Axiomatic Classification Tree Analysis (ACT) method, where an axiomatic model of program complexity was used to develop the candidate measures in the classification tree.

In all important measures, the ACT was either as good as or improved upon the original CTA model: (1) Classification trees were smaller; (2) Coverage was the same; (3) Accuracy improved; (4) Consistency improved and (5) Completeness was the same. We therefore believe that we have a candidate process that improves upon the original model.

Using an axiomatic basis for classification trees has two important economic benefits:

1. By eliminating unnecessary measures from the classificaiton tree (e.g., reducing the list from 74 to 18 in the NASA SEL experiment), we eliminate the need to collect such data. This would imply less overhead on the development process.
2. The axiomatic classification tree analysis technique generates improved results, allowing management to better control and evaluate the development process and allow for more informed decision making with less risk involved.

Of course there is still much more to be done. ACT is only right on 50% of the modules it calls high cost, and only finds accurately over one third of these modules. However, the method is improving, and is inexpensive to use since it is available as a byproduct of static analysis of the developing code. Further work will continue on developing these models.

Acknowledgements

This research was supported in part by National Science Foundation grant CCR-8819793 and National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

References

[1] Cárdenas S. and M. V. Zelkowitz, "A management tool for the evaluation of software designs," *IEEE Trans. on Software Engineering* 17, 9 (September, 1991) 961-971.

[2] Porter A. A. and R. W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees", *IEEE Software*, (March, 1990) 46-54.

[3] Selby R. W. and A. A. Porter, "Learning from example: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. on Software Engineering* 14, 12 (1990) 1743-1757.

[4] Tian J. and M. V. Zelkowitz, "A formal program complexity model and its application," *J. of Systems and Software* 17, 3 (March, 1992) 253-266.

[5] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. on Software Engineering*, 14, 9 (1988) 1357-1365.

Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development

Lionel C. Briand, Victor R. Basili and Christopher J. Hetmanski

**Institute for Advanced Computer Studies,
Computer Science Department,
University of Maryland, College Park, MD, 20742**

To be published in the proceedings of the IEEE International Symposium on Software Reliability Engineering, North-Carolina, USA, October 1992.

Abstract:

Applying equal testing and verification effort to all parts of a software system is not very efficient, especially when resources are limited and scheduling is tight. Therefore, one needs to be able to differentiate low / high fault density components so that testing / verification effort can be concentrated where needed. Such a strategy is expected to detect more faults and thus improve the resulting reliability of the overall system. This paper presents an alternative approach for constructing such models that is intended to fulfill specific software engineering needs, (i.e. dealing with partial / incomplete information and creating models that are easy to interpret). Our approach to classification is to (1) measure the software system to be considered and (2) build multivariate stochastic models for prediction. We present experimental results obtained by classifying FORTRAN components developed at the NASA Goddard Space Flight Center into two fault density classes: low and high. Also, we evaluate the accuracy of the model and the insights it provides into the software process.

Key words: fault-prone software components, stochastic modeling, machine learning.

1. Introduction

In this paper, we address the issue of identifying high fault density software components via empirical stochastic modeling. If we can identify components that produce a great deal of faults relative to their size, then we can concentrate the verification and testing processes on them and thereby optimize the resulting reliability of the developed software system. However, building such stochastic models is a difficult task. The data collected is often incomplete and/or heterogeneous and presents many problems with respect to model construction (e.g. interdependencies, outliers, complex relationships). In this paper, we present an alternative modeling process based on both statistics and machine learning principles [M83]. We show how the process facilitates the identification of high fault density components based on metrics obtainable at the end of the coding phase.

The modeling approach presented in this paper, called Optimized Set Reduction (OSR), has been developed at the University of Maryland [BBT91] in the framework of the TAME project [BR88] . It is derived from the ID3 model [Q79, Q86, BR84] which was originally developed for automatic generation of classification/decision trees. As discussed in [CE87,BBT91], the use of ID3 has several inherent problems and leaves room for improvement with respect to many data analysis and modeling issues (i.e. small data sets, missing data values, noisy data, heteroscedasticity). Our motivation for developing OSR and a tool to support it was to design a data analysis technique matching, to the extent possible, the specific needs of building multivariate empirical models for software engineering. The issue of using OSR for predicting on a continuous range is addressed in [BBT91]. In this paper, we discuss using OSR to classify software components into two fault density classes (low, high).

In Section 2, we present the basic principles of the OSR algorithm and formally define the approach. This formalism is intended to give an unambiguous presentation of some of the features of OSR rather than a complete definition of it. Section 3 discusses the issue of building models based on partial information (i.e. missing data for technical or cost reasons). Section 4 presents a process called "pattern merging" whose goal is to facilitate interpretation and learning based on the generated models. Sections 5 and 6 present some of the results obtained via

Research this study was supported in part by NASA grant NSG 5123 and by AFOSR 90-0031

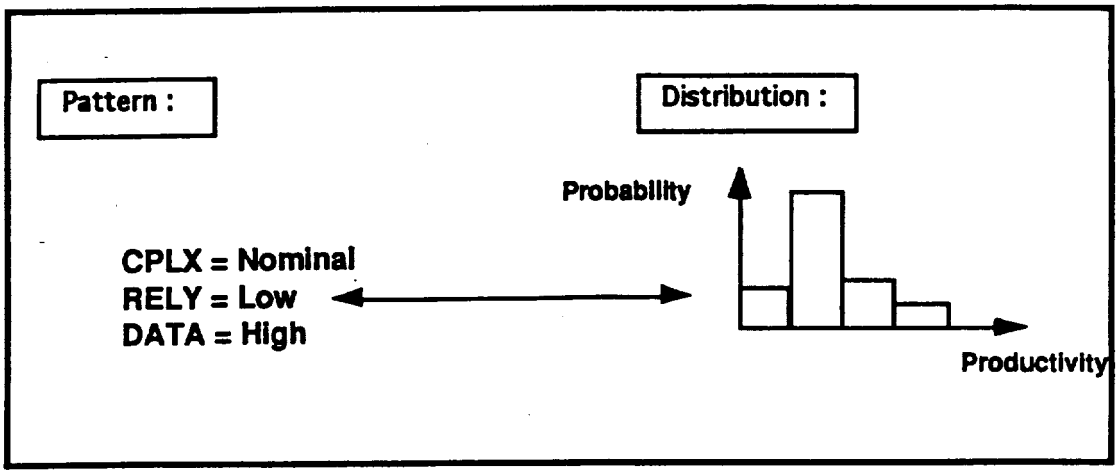


Figure 1: Example of a Pattern and its Associated Probability Distribution

experimentation using OSR. Based on these results, we can determine the accuracy of the model. Also, we can compare OSR's outputs with those of a *logistic regression* based model, which is one of the most standard statistical techniques for classification [HL89, AG90]. Finally. Section 7 underlines the major conclusions and directions for future research.

2. Optimized Set Reduction

2.1 Basic Principles

Let us assume we want to assess a particular characteristic of an object (e.g. the fault density of a component). We will refer to this characteristic as the Dependent Variable (Y). The object is represented by a set of explanatory variables which describe the software component (called Xs). These variables can be either discrete or continuous. For example, a software component may be described by two Xs, its cyclomatic complexity (continuous) and the type of its function (discrete). Also, assume we have a historical data set containing a set of *pattern vectors* that contain the previously cited Xs plus an associated actual Y value. We will call the Xs portion of the pattern vector a *measurement vector*.

The goal of the OSR algorithm is to determine which subsets of experiences (i.e. pattern vectors) from the historical data set provide the best characterizations of the object to be assessed. In other words, we try to determine which subsets of the data set yield the "best" probability distributions on the Y range. A good probability distribution is a probability distribution concentrating a large number of pattern vectors in either a small part of the range (Y is continuous) or in a small number of dependent variable categories (Y is discrete). One of the commonly used probability distribution evaluation functions is the *information theory entropy* (H). Alternative probability distribution evaluation functions are discussed in [Q86, SP88, M89]. Each of the subsets of the historical data set yielding "optimal" distributions, referred to as *optimal subsets*, are characterized by a set of conditions (referred to as *predicates*) which are true for all pattern vectors in that subset. Each set of predicates characterizing a subset is called a *pattern*. Figure 1 shows an example of a pattern and its associated probability distribution in the data set. The pattern is composed of three predicates where the dependent variable to be assessed is "development productivity". Figure 1 shows that if these predicates (i.e. ComPLeXity = Nominal, RELiabilitY=Low, DATA base size = High) are true for a project, then its productivity is most likely to be in the second productivity class.

2.2 Formal Definition of the OSR Process

We want to identify *optimal* subsets in the historical data set. We can formalize the process using set theory and predicate calculus by defining the function Opt. Let us assume we have a set of m explanatory variables $\{X_1, X_2, \ldots, X_m\}$ and a corresponding set of explanatory variable value domains $\{EV_1, EV_2, \ldots, EV_m\}$. Let us define the measurement vector domain to be $MV = \underset{i=1..m}{\times} EV_i$. The dependent variable value domain (DV) may be seen as a set of classes which can be either intervals or categories. Therefore, the value domain of the pattern vectors in the data set can be represented as $PV = DV \times MV$. Let PVS be a set of pattern vectors representing the historical data set ($PVS \subseteq PV$). A predicate is a variable value pair (i.e. an X_i and its corresponding explanatory variable value).

• Definition 1: Let PSS be a subset of PVS and let the measurement vector mv describe the object to assessed. VALID(PSS, mv) is true if mv is composed by at least one predicate which is true for all the pattern vectors in the set PSS.

$$PSS \subseteq PVS \wedge mv \in MV \wedge \exists i \in (1 .. m)$$
such that $\forall pv \in PSS\ (mv(i) = pv(i))$
$\Rightarrow$ VALID(PSS, mv)

• Definition 2: TC(PSS, PVS) is true if the two data sets PSS and PVS do not show a statistically significant difference in distribution on the DV range. This is may be evaluated by performing statistical inference tests for comparing distributions. We currently use a binomial test for proportions since it does not have any applicable restraints (e.g. minimum expected frequencies like the Chi-squared test of independence)[CA88]. For each dependent variable class, the probability that proportions in PSS and PVS differ by chance is calculated. If for at least one of the classes, this probability is below a level of significance TC defined by the user, then we reject the hypothesis that the two distributions are identical. TC stands for *Termination Criterion* because the OSR process will be terminated if the condition defined by TC is true.

• Definition 3: EMIN(PSS_1, PVS) is true if PSS_1 is one of the subsets of PVS yielding a minimal *normalized entropy* H upon all statistically significant subsets of pattern vectors (e.g. a one vector subset has a minimal entropy but it is not a statistically significant subset and therefore is not relevant here).

$$(PSS_1 \subseteq PVS \wedge \neg TC(PSS_1, PVS))$$
$$\wedge (\forall PSS_2 \subseteq PVS\ (\neg TC(PSS_2, PVS) \wedge H(PSS_1) \leq H(PSS_2)))$$
$$\Rightarrow EMIN(PSS_1)$$

where
$$H(PSS) = \sum_{d \in DV} - p(PSS,d) \log_{1000} p(PSS,d)$$
where
p(PSS, d) is the a priori probability that a vector which is an element of PSS has a dependent variable value belonging to the dependent variable class d

• Definition 4: Opt(PVS, mv) is a function yielding a set of *optimal pattern vector subsets.*

$$Opt(PVS, mv) = \{PSS \subseteq PVS \mid VALID(PSS, mv) \wedge EMIN(PSS, PVS)\}$$

However, the function Opt as defined cannot be used as an algorithm to extract the optimal subsets. The most important reasons are:

- The number of possible predicate combinations makes the search execution time prohibitive.
- We want the patterns to contain a minimal set of predicates, i.e., we want all the predicates in the pattern to have a significant impact on the resulting pattern entropy.
- We loose some information about the relative impact of the various predicates in the entropy reduction process.
- The contexts in which the various predicates appear relevant are undetermined.

Therefore, we implement a *greedy* algorithm using the function Opt which addresses the issues mentioned above. The Optimized Set Reduction algorithm can be roughly described by a three step recursive algorithm.

• Step 1: If the dependent variable is continuous, its range is divided into a set of classes according to two main factors: the necessary model accuracy and the size of the data set. Then, the ranges / categories of the explanatory variables are divided / clustered into *classes* (e.g. $Class_{i1}$ ... $Class_{ij}$ for the explanatory variable X_i) based on meaningful class creation techniques. For example, a Complexity range can be divided in three classes: low, average, high. Numerous techniques can be used in order to create meaningful classes (e.g. cluster analysis) [DG84]. However, this issue will not be addressed in this paper.

• Step 2: Select all the pattern vectors in the data set having a value for the explanatory variable X_i belonging to $Class_{ik}$, where the X_i for the object to be assessed belongs to the same class, and where the subset characterized by the predicate $X_i \in Class_{ik}$ yields the minimum statistically significant value for H. However, several subsets (characterized by different predicates) yielding "similar" minimal entropies (i.e. the similarity criterion has to be defined by the user of the algorithm) can be extracted at once. Let us call PSS_i the extracted subsets of pattern vectors.

• Step 3: Step 2 is repeated in a recursive manner on each subset PSS_i and each successive subset until the user defined termination criteria (TC) is reached.

This OSR algorithm can be formally specified as a two parameter recursive function where PVS is the historical data set and mv the vector describing the object to be assessed:

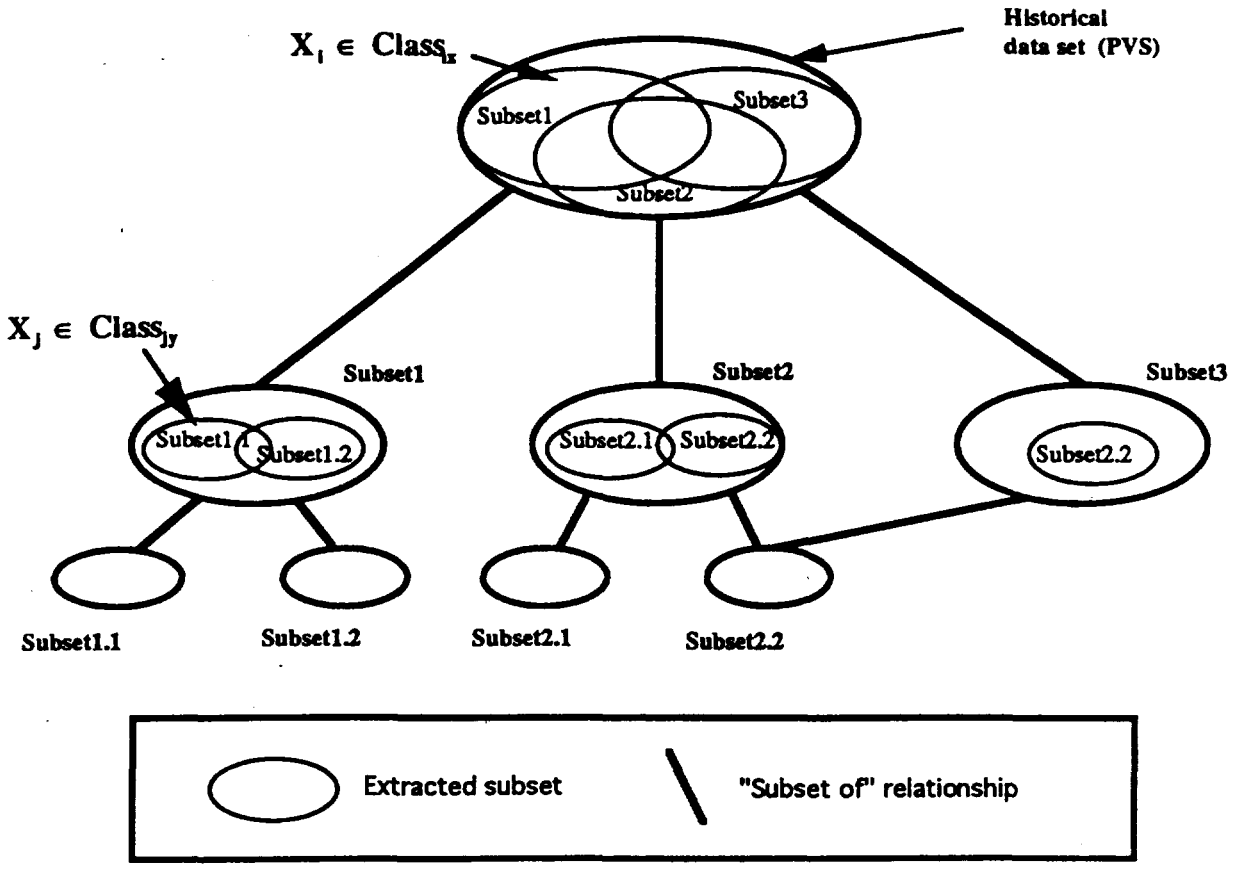


Figure 2: Example of OSR Hierarchy

$$OSR(PVS, mv) = \text{if } Opt(PVS, mv) \neq \emptyset \text{ then } \bigcup_{PSS \in Opt(PVS, mv)} (OSR(PSS, mv)) \text{ else } \{PVS\}$$

The whole subset extraction process can be represented as a hierarchy (see Figure 2). Note that this representation should not be confused with a partition tree since: (1) the extracted subsets are not exclusive and (2) a subset can have several parent subsets. Each path of the hierarchy represents a generated pattern (e.g. Figure 2: $X_i \in Class_{ix}$ AND $X_j \in Class_{jy}$ defines Subset1.1) which is relevant to the particular prediction to be performed. As shown in Figure 2, two patterns may yield exactly the same subset (e.g. Subset 2.2). The extracted subsets (i.e. leaves of the hierarchy) which form various probability distributions across the dependent variable range may show different trends. For each leaf probability distribution, if the dependent variable is discrete, the dependent variable class containing the largest number of pattern vectors may be selected as the most likely class for the new object (characterized by mv) to lie in. Using an alternative *Bayesian approach*, a *loss / risk function* could be defined by the user [BBT91]. In this case, the dependent variable class yielding the minimum *expected loss* is selected. Each pattern prediction (i.e. hierarchy leaf) is used to make a final global prediction based on predefined decision rules. In order to perform such decisions effectively, we need to be able to evaluate the accuracy of the identified patterns. This issue is treated in Section 3.

3. Handling Partial Information with OSR

3.1 Definition of the Problem

As mentioned above, analyzing complex data sets and variable relationships is a very difficult task for several reasons (i.e. incomplete / heterogeneous / small data sets, missing data, complex interdependencies). The most common of these is the problem of partial information. Our lack of understanding of software processes (due to our lack of experience and the wide variability from one development environment to another) makes experience difficult to reuse. Also, because of cost and schedule related constraints, necessary data cannot always be collected. All of these issues contribute to the incompleteness of our data.

Missing information reduces our ability to predict and understand. However, we have to establish whether or not the lack of a piece of data is an obstacle to prediction. This means that we need a model that both generates predictions and provides some insight into the reliability of each individual prediction. A goodness indication at the model level such as the coefficient of determination in least-squares regression analysis is not sufficient since it fails to yield an individual reliability measure for each prediction.

For example, let us say we wish to predict project productivity according to collected physical features of the system and predefined quality requirements. Suppose we do not have any information about the team experience related to the programming environment and the application domain. This information might be somewhat irrelevant, i.e. if the structural complexity of the software and the required system reliability are low, then the variance of the prediction is small. However, if high reliability on a complex software system is expected, then people rated as having low experience are likely to generate schedule and/or budget slippages. This will make any prediction based exclusively on other criteria meaningless. Therefore, we need a modeling approach that can answer the question: Do I have enough information to make a reliable prediction?

3.2 Solutions to Partial Information within the OSR Framework

For each measurement vector in the historical data set we run the OSR algorithm using as an initial data set (i.e. set at the top of the OSR hierarchy) the historical data set minus the measurement vector to be predicted. It is removed from the data set in order to avoid any bias in the results. We therefore extract specific patterns for each measurement vector and form a set of patterns representing the trends observable on this particular data set.

This resulting set of patterns, or Specific Pattern Set (SPS) may be seen as a model of the historical data set. Many of these patterns will be the same or "similar" and will therefore form classes of patterns. For each of these classes, based on the SPS, we can evaluate statistics such as *pattern reliability* (i.e. percentage of correct classification) or *pattern significance* (i.e. the probability that the reliability is greater than or equal to the one observed by chance) by comparing the predicted DV values with the actual ones. These statistics can then be used to evaluate predictions as explained in the subsequent paragraphs. The process of generating a SPS will be referred as to *Development Environment Analysis* (DEA).

In the text below, we assume the produced patterns have the following conjunctive normal form:

Predicate1 AND Predicate2 AND ... AND PredicateN

However, a pattern is not only a logical proposition. The order in which the predicates appear in the hierarchy (Figure 2) is relevant from an understanding perspective. A predicate is relevant only when the conditions defined by its preceding / parent predicates in the hierarchy (i.e. referred as to the *context* of a predicate in a particular pattern) are true. For example, Predicate 1 significantly reduces entropy by itself. Also, in the context of Predicate1, Predicate2 significantly reduces entropy. However, based on this pattern, there is no evidence that Predicate2 significantly reduces entropy by itself.

The notion of pattern reliability and significance, as mentioned above, can be more formally defined as follows: the reliability of a pattern with respect to a particular dependent variable class is the probability that the pattern will predict the correct value for the dependent variable.

Let $DVclass_i$ be dependent variable class i. Let T equal the number of generated patterns $\{P_j\}$ that predict $DVclass_i$. Let C equal the number of patterns which correctly predict $DVclass_i$ (based on the actual DV value of the pattern vector for which the pattern was produced during DEA).

Then we define the *reliability* of P_j with respect to the dependent variable class $DVclass_i$ as:

$$R\,[DVclass_i\,;\,P_j] = C / T$$

The probability that a pattern appears T times yielding a particular classification $DVclass_i$ C times correctly by chance (P(C,T,p)) can be expressed by the binomial distribution:

$$P(C,T,p)=\frac{T!}{C!(T-C)!}p^{C}(1-p)^{T-C}$$

where, $p = p(DVclass_i)$, i.e. the a priori probability that the value of the dependent variable is in $DVclass_i$.

If the pattern reliability R is equal to 1.0, then the binomial equation can be simplified and the level of significance is simply p^T. If R is below one, then the *pattern significance* S can be calculated by using the following formula:

$$S=\sum_{i=0}^{T-C}P(C+i;T;p)$$

Since we are able to differentiate *significant, reliable* patterns from the non-significant and/or unreliable ones, we can assess the reliability of the prediction when we make it. A prediction based on a reliable pattern with a sufficient level of significance (e.g. $S < 0.05$) is

believable, whereas, one based on a reliable pattern with a poor level of significance is not. A poor reliability means that a pattern is not robust to "noise" (i.e. the dependent variable variations created by unknown or non-measured explanatory variables). A poor significance may mean that the pattern is a result of noise or more complex phenomena which are beyond the scope of this paper.

4. A Process for Merging Patterns

Patterns are useful both for predicting variables of interest (e.g. fault density) and providing understandable / interpretable models. However, interpreting the patterns generated by a DEA would force the user to deal with useless complexity. Many of these patterns are similar and should not be differentiated. This can prevent the user from getting a clear picture of the model trends. Therefore, the patterns generated by the OSR process need to be grouped in order to make them more easily understandable and interpretable. This can be done using a formally defined statistical process (described below) where the user fixes the desired level of "similarity" between pattern by assigning values to a small set of parameters.

Let us define two patterns PT1 and PT2:

PT1: $X_j \in Class_{jy}$ AND $X_i \in Class_{ix}$
PT2: $X_j \in Class_{jy}$ AND $X_k \in Class_{kt}$

Suppose in the context where $X_j \in Class_{jy}$, the pattern vector set for which $X_i \in Class_{ix}$ happens to show a strong *association* with the one for which $X_k \in Class_{kt}$. This implies that these predicates capture basically the same phenomenon. The strength of the association can be assessed by using normalized Chi-squared based statistic such as Pearson's Phi [CA88]. A Chi-squared test can be performed in order to assess the statistical level of significance of such an association. The two patterns will be merged into one signifying that the selection of one predicate, or the other, during the OSR process, occurs by random. This is a result of slight differences between the two predicates and therefore distinguishing between them does not help to understand the object of study. This phenomenon is mainly due to complex interdependencies between Xs that are often underlying the software engineering data sets.

The notion of a "slight difference" is rather subjective and therefore must be defined by the user. Thus, he / she declares either a Phi value (actually Phi ^{2} which better represents in this case the degree of association [CAP88]) or a level of significance which represents the minimal degree of association necessary to assume two predicates as similar. This process of *merging patterns* based on the *similar predicates principle* yields the resulting pattern PT{1,2} which contains the *composite predicate* ($X_i \in Class_{ik}$ OR $X_k \in Class_{kt}$), implicitly meaning that its two component predicates are interchangeable in this context.

PT{1,2}: $X_j \in Class_{jy}$ AND ($X_i \in Class_{ix}$ OR $X_k \in Class_{kt}$)

Automated merging of similar patterns can be performed if the user provides either a Phi value or a level of significance that would correspond to an unambiguous definition of *pattern similarity*.

In a similar manner, we can define a second merging principle. Let us suppose we have the following patterns:

PT1: $X_j \in Class_{jy}$ AND $X_i \in Class_{ix}$
PT2: $X_j \in Class_{jy}$ AND $X_i \in Class_{it}$

Let us assume that $Class_{ix}$ is a neighbor class of $Class_{it}$ on the X_i range. In this particular case, if the two patterns characterize subsets with no statistically significant difference in distribution on the DV range, then they can be merged. This is because the variation from one class to the other seems to have a non-relevant effect on the dependent variable in the context where $X_j \in Class_{jy}$. Therefore, in order to assess if merging is possible, the probability that differences between distributions are due to random is calculated. For each dependent variable class, the proportions of pattern vectors are compared between the two distributions by calculating the probability that difference in proportion is due to random. If for all dependent variable classes, the resulting minimum probability is above a user-defined critical probability value, we accept the hypothesis that there is no significant difference between the two distributions. In the current tool, this is calculated through a binomial test in order to avoid the assumptions related to other more computationally effective tests (e.g. Chi-squared test of independence) [CAP88].

Both of the merging principles defined above can be used simultaneously in order to obtain more general patterns. However, the merging process using both of them must be carefully defined. In a tool, such mechanisms can be completely automated. The user would have to define some thresholds / criteria allowing the algorithm to declare two predicates *similar* (i.e., a level of significance, Phi value) and/or two classes *similar* (i.e., critical probability value). Before the merging process starts, the tool will calculate the matrix containing all the phi values and levels of significance between all predicates. Then, the merging process for the first position predicates starts: it is a several pass process where only two predicates can be merged at a time. First, predicates are merged according to the *similar class principle*. Then, the pairs of predicates

showing the strongest significant associations are merged (*similar predicate principle*). During the next passes, predicates can be merged to composite predicates and composite predicates to composite predicates. The process stops when no merging is possible according to the criteria defined by the user. Once finished, association matrices are calculated within the contexts defined by each unique first position predicate (composite or not) resulting from the first pass. Then, the merging process for second position predicates begins within each context following the rules defined above. This is repeated successively on increasing predicate positions until a predefined (i.e. by the user) *maximum merging level* is reached. Thus, the user defines the number of predicate positions he / she wants to look at and therefore set the maximum merging depth of the algorithm.

5. Experiment Design

Our goal in this article is to describe a technique to distinguish between low and high risk components.

The notion of risk has multiple dimensions. We focus here on the identification of low/high fault density components. If we can distinguish between these two types of components, then we can concentrate on the high fault density ones during the verification and testing process. Moreover, if we can build this kind of model for each kind of fault, we can apply fault specific testing techniques to localize and correct faults. Basili and Selby showed in [BS87] that the effectiveness of three of the most well known testing approaches could vary significantly according to the type of fault considered. Although more experiments are needed to better understand the issue, this study supports the idea of building different models for each type of fault.

The collected data set is based on fifteen FORTRAN projects which were developed at the NASA Goddard Space Flight Center in the early eighties. On all of these project, static measures at the component level were collected using a static code analyzer. Fault report forms were filled out during the test phases of the development process. Faults were identified, classified according to a predefined taxonomy and localized in the system.

Our definition of fault density is the ratio of the number of faults over the number of executable statements. In this experiment we will look, as a first step, to faults related to incorrect data structure readings or writings (called "data value" faults in the NASA Software Engineering Laboratory). This type of fault represents about 50 percent of the total number of faults collected on the projects studied in this experiment.

6. Experimental Results

6.1 Prediction Results

We used the OSR technique to build classification models that were intended to provide an answer to the question: Is this component likely to be in the lowest / highest quartiles on the "data value" fault density range? This was done by performing a DEA on the data set which contained 399 pattern vectors. Each pattern vector was comprised of a list of static measures which describe a software component (i.e. the measurement vector), plus, the fault density of that component. Thereby, we were able to calculate an average classification correctness (i.e. percentage of components correctly classified) of the OSR model . Also, we try to demonstrate through examples that reliable patterns can be differentiated from misleading patterns.

For the sake of simplicity, we will look only at the two first predicates (the most relevant according to the OSR selection mechanism) of each of the generated patterns. R, O and S are respectively the Reliability, number of Occurrence (the number of times a pattern appeared), and the Significance of the pattern. The explanatory variable ranges were divided into quartiles. This method is the simplest technique for class creation but most likely the least effective. The class creation process is one of the issues that remains to be investigated (See Conclusion). OSR suggested that low and high fault density components were partly characterized by the following significant (< 0.05 level of significance) and non-significant patterns:

Low Fault Density Components

Assume that Fq, Sq, Tq and Lq represent respectively the First quartile, Second quartile and so forth, on the explanatory variable ranges.

• Examples of Highly-Significant Reliable Patterns:

PT1: # stmts $\in$ Lq AND # calls $\in$ Fq,
R = 1.0, O = 18, S = 0.000

PT2: # stmts $\in$ Lq AND # calls $\in$ Sq,
R = 1.0, O = 17, S = 0.000

PT3: # stmts $\in$ Lq AND # format/stmt $\in$ Fq,
R = 1.0, O = 10, S = 0.000

PT4: # stmts $\in$ Lq AND # i/o stmt / stmt $\in$ Fq,
R = 1.0, O = 15, S = 0.000

PT5: # stmts $\in$ Lq AND # assign/stmt $\in$ Fq,
R = 1.0, O = 8, S = 0.004

PT6: # stmts $\in$ Lq AND # decis_node/stmt $\in$ Fq,
R = 1.0, O = 11, S = 0.005

PT7: # stmts $\in$ Lq AND #funct/stmt $\in$ Tq
R = 1.0, O = 24, S = 0.000

PT8: # decision nodes $\in$ Lq AND # calls $\in$ Fq,
R = 1.0, O = 14, S = 0.000

PT9: # decision nodes $\in$ Lq AND # calls $\in$ Sq,
R = 1.0, O = 15, S = 0.000

PT10: # decision nodes $\in$ Lq AND # i/o stmts $\in$ Fq,

R = 1.0, O = 11, S = 0.001

PT11: # operators/stmt ∈ Fq AND # calls ∈ Fq,
R = 1.0, O = 9, S = 0.002
PT12: # operators/stmt ∈ Fq AND # format/stmt ∈ Fq,
R = 1.0, O = 6, S = 0.016
PT13: # operators/stmt ∈ Fq AND # functions ∈ Lq,
R = 1.0, O = 8, S = 0.004

• Examples of Non-Significant Reliable Patterns

PT14: # stmts ∈ Tq AND # format/stmt ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT15: # stmts ∈ Tq AND # i/o stmt/stmt ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT16: # stmts ∈ Tq AND # i/o stmts ∈ Fq,
R = 1.0, O = 2, S = 0.25
PT17: # stmts ∈ Tq AND # i/o stmts ∈ Sq,
R = 1.0, O = 4, S = 0.0625
PT18: # operators/stmt ∈ Fq AND # funct/stmt ∈ Lq,
R = 1.0, O = 4, S = 0.0625

• Example of a Non-Significant Non-Reliable Pattern

PT19: # stmts ∈ Tq AND # functions ∈ Tq,
R = 0.0, O = 1, S = 1.000

High Fault Density Components

• Examples of Significant Reliable Patterns

PT1: # lines ∈ Fq AND # comment/stmt ∈ Tq,
R = 1.0, O = 11, S = 0.001
PT2: # stmts ∈ Fq AND # comment/stmt ∈ Tq,
R = 0.94, O = 17, S = 0.000
PT3: # format/stmt ∈ Lq AND # comment/stmt ∈ Tq,
R = 1.0, O = 10, S = 0.001
PT4: # decisions nodes ∈ Fq AND # call/stmt ∈ Lq,
R = 0.95, O = 21, S = 0.000
PT5: # stmts ∈ Fq AND # calls ∈ Sq,
R = 0.94, O = 18, S = 0.000
PT6: # stmts ∈ Fq AND # i/o stmt/stmt ∈ Sq,
R = 1.00, O = 13, S = 0.000
PT7: # stmts ∈ Fq AND # operand/line ∈ Sq,
R = 1.00, O = 20, S = 0.000
PT8: # stmts ∈ Fq AND # operand/stmt ∈ Sq,
R = 1.00, O = 18, S = 0.000
PT9: # stmts ∈ Fq AND # i/o variable/line ∈ Fq,
R = 1.00, O = 27, S = 0.000
PT10: # stmts ∈ Fq AND # operators ∈ Sq,
R = 0.91, O = 11, S = 0.006
PT11: # operator/stmt ∈ Lq AND # assign/stmt ∈ Lq,
R = 1.0, O = 6, S = 0.015

As shown in the above results, significant reliable patterns can be recognized and differentiated from the non-reliable / non-significant ones. Therefore, significant reliable patterns can be identified and used with confidence for both prediction and interpretation. For instance, if we take pattern PT1 for low density components, we observe a reliability of 100% based on 18 occurrences. This produces a very good pattern significance. The predictions generated by this pattern can therefore be considered very reliable and used with confidence. Both the OSR patterns and the logistic regression model yield an average classification correctness of 82%. This result is very encouraging considering that the class creation process used (i.e. dividing the range in quartiles) was primitive and that the explanatory variables available are all continuous (which is an important advantage for the logistic regression model). Moreover, note that the OSR process is entirely automated.

The patterns produced by OSR are not always easy to interpret. Interpretation of patterns (or any other stochastic model) requires expert knowledge. However, in the next subsections, we provide some rules for reading and interpreting the above patterns. Some pattern merging results are also provided.

6.2 Pattern Interpretation Rules

Interpretation of patterns is much easier than interpreting regression coefficients. First, OSR takes into account the fact that an explanatory variable can have a strong impact in a certain context (defined by the predicates in preceding positions) and not be relevant in another one. Second, if strong associations exist in a given context, then the pattern merging process makes it apparent by creating composite predicates (see examples in section 6.3). The variation of reliability generated by a particular predicate can help assess the significance of the impact of an explanatory variable (on the dependent variable) when the explanatory variable belongs to a certain class of values within a certain context. Let us take the following pattern as an example: #stmts ∈ Lq AND #calls ∈ Fq which yields a reliability of 100%. However, #stmts ∈ Lq alone only yields a reliability of 88%.

This result suggests that #calls ∈ Fq is a relevant predicate in the context where #stmts ∈ Lq because it shows a significant impact on the fault density.

However, a pattern must always be interpreted in context. In some contexts (e.g. #stmts ∈ Fq), a variable (e.g. #operators) may not take on the full range of values. The interpretation of patterns like pattern PT10 for high density components must be done carefully: #operators ∈ Sq may be interpreted as a "rather large" number of operators because in the context #stmts ∈ Fq, very few components show either #operators ∈ Tq or #operators ∈ Lq (i.e. # stmts is strongly associated with # operators). Therefore, the OSR process did not select patterns like #stmts ∈ Fq AND #operators ∈ Tq since they yielded subsets that met the termination criteria. This example shows that even though interpreting patterns is always

simple, it requires the support of a tool .

6.3 Pattern Merging Results and Interpretation of Recognized Patterns

In this section, we intend to show how the merging process can help to group similar raw patterns into composite patterns and therefore provide more easily interpretable information. If we simplify the raw patterns generated by OSR using the merging criteria: Phi $^2 = 0.40$ and critical probability value of 0.0005, we get a set of composite patterns for each of the dependent variable classes. In order to illustrate the point, we first show some of the intermediate steps of the merging process. Then we give two composite patterns: CP1 and CP2 (formed by the merging process), which characterize low fault density components.

For example, low density component patterns PT1 and PT2 can be merged based on the similar classes principle. They both show the same first predicate: # stmts $\in$ Lq. Their second position predicate shows the same variable # calls and two neighboring classes (Fq and Sq). Since they do not show a statistically significant difference is distribution (critical probability value = 0.0005), then they can be merged in: #stmts $\in$ Lq AND # calls < MEDIAN.

Similarly, low density component patterns PT3 and PT4 can be merged based on the similar predicate principle. They both show the same first position predicate and their second position predicates are strongly associated (Phi^2 = 0.57). Therefore, they can be merged in: #stmts $\in$ Lq AND (#formats/stmt $\in$ Fq OR #I/O stmts/stmt $\in$ Fq).

This merging process is repeated until no more merging is possible according to the user's criteria. CP1 and CP2 are the final resulting composite patterns which characterize low fault density components:

CP1: SIZE_HIGH AND CALLS & I/O_LOW,
R = 99% , O = 169, S = 0.000

CP2: SIZE_HIGH AND FUNCT_HIGH,
R = 86%, O = 43, S = 0.000

where the composite predicate SIZE_HIGH is defined as:

$$\text{SIZE\_HIGH} \Leftrightarrow \begin{pmatrix} \#\text{ statements} \in \text{Fq OR } \#\text{ statements} \in \text{Sq} \\ \text{OR } \#\text{ formats} \in \text{Lq OR } \#\text{ decision nodes} \in \text{Lq} \\ \text{OR } \#\text{ operators / stmt} \in \text{Fq} \end{pmatrix}$$

and, in the context where SIZE_HIGH is true, the following composite predicates are formed:

$$\text{CALLS \& I/O\_LOW} \Leftrightarrow \begin{pmatrix} \#\text{ calls} \in \text{Fq OR } \#\text{ calls} \in \text{Sq} \\ \text{OR I/O stmts/stmt} \in \text{Fq OR } \#\text{ formats/stmt} \in \text{Fq} \\ \text{OR } \#\text{ I/O stmts} \in \text{Fq OR } \#\text{ I/O stmts} \in \text{Sq} \end{pmatrix}$$

$$\text{FUNCT\_HIGH} \Leftrightarrow \begin{pmatrix} \#\text{ functions} \in \text{Tq OR } \#\text{ functions} \in \text{Lq} \\ \text{OR functions/stmt} \in \text{Tq OR functions/stmt} \in \text{Lq} \end{pmatrix}$$

CP1 and CP2 actually define classes of raw patterns that are assessed equivalent according to the user-defined criteria. Some of the low density patterns presented in section 6.1 belong to CP1: PT1, PT2, PT3, PT4, PT8, PT9, PT10, PT11, PT12, PT14, PT15, PT16, PT17 and others to CP2: PT7, PT13, PT18, PT19. Both of the composite patterns suggest that large components are likely to have low fault densities. This agrees with a study conducted by Basili and Perricone [BP84]. This may be partially explained by the fact that low operator densities seem to be strongly associated with large components. CP1 suggests that a low number of function calls or a low number of I/O statements increase the probability of having a low fault density. CP2 indicates that a large component showing a high density of functions is likely to show a low fault density.

Merging patterns is <u>always</u> desirable. It allows us to combine related, rare, isolated patterns to more significant patterns and thereby group together trends which capture essentially the same phenomenon. This makes the generated composite patterns easier to interpret and gives the user a more abstract and general view of the results. Also, as we have seen, patterns with a small number of occurrences cannot be trusted (even though they show good reliabilities) because of their weak level of significance. However, if these patterns are shown to be strongly associated with other reliable patterns, then the significance of the generated composite pattern increases. This allows us to gain more trust in rare reliable patterns based on the calculated composite pattern's level of significance. However, this should be used very carefully and needs further investigation.

7. Conclusion

Based on the above experimental results, building useful models for assessing the fault density of software components, based upon early available simple metrics in the presence of noisy data appears possible. Whenever OSR generates a very reliable and significant pattern, the prediction can be used with confidence. To the contrary, if the pattern is not a reliable and significant one, an alternative modeling method such as logistic regression may give a more believable prediction. We have seen that problems such as partial information in the data set can be accommodated for by assigning a relative goodness to each prediction. Also, the patterns appear to be easier to interpret than regression coefficients and correlation

matrices which are the usual outputs of regression analysis. This is due mainly to the fact that OSR produces symbolic / logical expressions where the notion of context is introduced by considering the order of the predicates. Also, the merging process helps the user look at the model at various level of abstraction. From a more general perspective, based on previous [BBT91, BP92] and current experimental results, OSR is a data analysis framework that successfully integrates statistical and machine learning approaches in empirical modeling with respect to specific software engineering needs. However, while the experimental results thus far have been encouraging, many aspects of the processes involved in OSR are still to be optimized. Such processes include, by order of importance, EV class definition, the refinement and automation of the merging process, support for pattern interpretation, the attribute selection process and the selection of termination criteria.

8 Acknowledgments

We thank Gianluigi Caldiera, Denis Oberkampf, William Thomas and especially Sandro Morasca for their excellent suggestions. We also thank the referees for their insightful comments.

References

[AG90] Alan Agresti, "Categorical Data Analysis", Wiley-interscience, 1990

[BP84] V. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, vol. 27, no. 1, January 1984.

[BR88] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-oriented Software Environments", IEEE Trans. Software Engineering 14 (6).

[BS87] V. Basili and R. Selby, " Comparing the Effectiveness of Software Testing Strategies", IEEE Trans. on Software Engineering 13 (12).

[BR84] L. Breiman et al, "Classification and Regression Trees", Wadworth & Brooks, 1984.

[BBT91] L. Briand, V. Basili and W. Thomas, "A Pattern Recognition Approach to Software Engineering Data Analysis", IEEE trans. Software Eng., Special issue on software measurement principles, techniques and environments, November 1992.

[BP92] L. Briand and A. Porter, "An Alternative Modeling Approach for Predicting Error Profiles in Ada Systems", European conference on quantitative evaluation of software and systems (EUROMETRICS'92), Brussels, Belgium, April 1992.

[CA88] J. Capon, "Statistics for the Social Sciences", Wadworth publishing company, 1988.

[CE87] J. Cendrowska, "PRISM: An Algorithm for Inducing Modular Rules", Journal of Man-Machine Studies, 27, pp.349.

[DG84] W. Dillon and M. Goldstein, "Multivariate Analysis", John Wiley & sons, 1984.

[HL89] D. Hosmer and S. Lemeshow, "Applied Logistic Regression", John Wiley & sons, 1989

[M83] R. Michalski, "Theory and Methodology of Inductive Learning." In R. Michalski, J. Carbonell & T. Mitchell (Eds.), Machine learning (Vol. 1). Los Altos, CA: Morgan Kaufmann.

[M89] J. Mingers, "An Empirical Comparison of Selection Measures for Decision-tree Induction", Machine learning 3, pp.319, 1989.

[Q79] J. Quinlan, "Discovering Rules by Induction from Large Collections of Examples", In D. Michie (Ed.), Expert System in the microelectronic age. Edinburg University Press, 1979.

[Q86] J. Quinlan, "Induction of Decision Trees", Machine learning 1, Number 1, pp.81, 1986.

[SP88] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision trees for Software Resource Analysis", IEEE trans. Software Eng., 1988.

Appendix: Definition of the Generalization Algorithm (notation consistent with section 2.2)

This generalization process can be formalized using the following definitions and algorithms:

• Definition A1: We define a composite predicate (cp) as $cp = \bigcup p,\ p \in PD$, which the set of all predicates. Composite predicates can be combined to form other composite predicates. Thus, we define $cp_{i \cup j} = cp_i \cup cp_j$.

• Definition A2: An *association coefficient* a_{ij}^{PSS} is an assigned statistical degree of association between cp_i and cp_j where PSS is the data set used to determine this association. Let us assume the two following data subsets:

$$PSS_i = \{pv \in PSS | cp_i \text{ is true}\}$$
$$PSS_j = \{pv \in PSS | cp_j \text{ is true}\}$$

A two row-two column contingency table is defined, where the subsets characterizing each row and column are respectively PSS_i, $PVS - PSS_i$, PSS_j, $PVS - PSS_j$. Based on this table, a Chi-Square based statistic (i.e. Pearson's Phi) defining the degree of association between the two subsets is calculated and assigned to a_{ij}^{PSS}.

• Definition A3: A *context* is a conjunction of a set of composite predicates that defines $PSS \subseteq PVS$. This defines the data subset on which an association coefficient is calculated and therefore its domain of validity.

• Definition A4: An *association matrix* $A_{n \times n}^{C}$ is a square matrix of association coefficients calculated in a context C, where the rows / columns represent all possible predicates.

example: $A_{n \times n}^{cp_k \wedge cp_l}$ contains all a_{ij}^{PSS}
where $\forall pv \in PSS$, $cp_k \wedge cp_l$ is true.

• Definition A5: Two composite predicates cp_i and cp_j are said to be *associated* in the context of C if $a_{ij}^{PSS} \geq$ some minimal level of association.This will be denoted as $cp_i \approx cp_j$.

• Definition A6: A *predicate tree* is a tree representation of the patterns generated during the Development Environment Analysis (i.e. DEA) process. As mentioned is Section 3.2, DEA produces a set of patterns representing the observed trends in the historical data set. It is expected that a significant number of these patterns will be duplicated or similar. This representation is a compact way of representing the specific pattern set (SPS). Each path of a predicate tree represent a pattern generated by DEA. (see Figure 3)

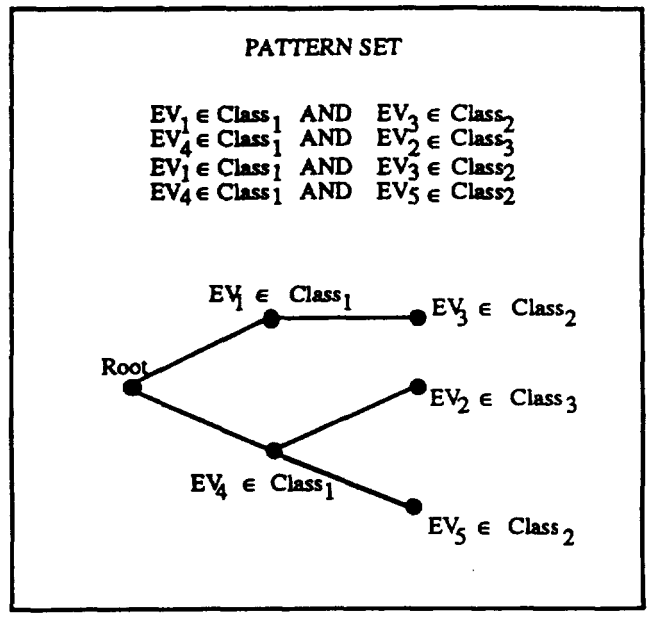


Figure 3: Example Predicate Tree

Notice that the root of the predicate tree is a "dummy" predicate which can be thought of as the *identity predicate* cp_I (i.e. $cp_i \wedge cp_I \Leftrightarrow cp_i$). Note that in the above example, all of the predicates are singleton. This represents a predicate tree before any generalization. Branches will be merged and composite predicates created at the nodes during the generalization process.

• Definition A7: The *maximum merging depth* (user defined) is the depth in the predicate tree to which generalization is to be performed. It defines the observation depth of the patterns by the user.

• Definition A8: Two composite predicates cp_i, cp_j are said to be "mergeable neighboring composite predicates" if the following conditions are fulfilled:

(1) There exist two predicates p_X: $Xi \in class_{ik}$, p_Y: $Xi \in Class_{it}$ such that p_X and p_Y are one of the disjunctive predicates of cp_i and cp_j, respectively.

(2) $Class_{ik}$ and $Class_{it}$ are neighboring classes on variable X_i range.

(3) cp_i and cp_j yield the same classification, show a difference of reliability below DR and a maximum

C-2

pattern level of significance S (i.e. DR and S are fixed by the user).

If these three conditions are true, then mncp(cp_i, cp_j, S, , DR) is true.

In order to define the generalization algorithm based on the above definitions, we assume that it starts with the procedure call: Generalize(predicate tree, root, cp_I, 0, PHI, DR)

We can now define the Generalize algorithm as follows:

procedure Generalize (predicate tree, node, context, current depth, PHI, DF)

(1) If the node is a terminal node of the predicate tree OR if depth > maximum merging depth then
RETURN

(2) while
$\exists$ cp_i, cp_j such that mncp (cp_i , cp_j, S, DF) do
merge(predicate tree, node, cp_i, cp_j)

(3) calculate $A_{mxm}^{context}$, the association matrix with all cp_i's, $i \in \{1,...,m\}$, in context.

(4) while $\exists$ cp_i, cp_j such that $cp_i \approx cp_j$ do

. select cp_i and cp_j such as $a_{i,j}^{context}$ is the strongest association in $A_{mxm}^{context}$
. merge(predicate tree, node, cp_i, cp_j)
. recalculate $A_{m-1 \times m-1}^{context}$, the association matrix for cp_i, ..., cp_{i-1}, cp_{i+1}, ..., cp_{j-1}, cp_{j+1}, ...,cp_m, cp_iUj in context.

(5) for each successor of node in predicate tree
Generalize (predicate tree, successor, context ^ cp_{node}, depth+1, PHI, DF)

end Generalize

In step (4), a call is made to procedure merge defined merge as follows:

procedure merge (predicate tree, node, cp_i, cp_j)
cp_i and cp_j are successors of node

(1) Combine cp_i and cp_j to form a single node cp_iUj

(2) Combine all like subpaths rooted at cp_iUj
end merge

A Classification Procedure for the Effective Management of Changes during the Maintenance Process

Lionel C. Briand and Victor R. Basili [1]

Computer Science Department and Institute for Advanced Computer Studies
University of Maryland
College Park, MD, 20742

To be published in the proceedings of the IEEE Conference on software maintenance, Orlando, Florida, USA, November 1992.

Abstract

During software operation, maintainers are often faced with numerous change requests. Given available resources such as effort and calendar time, changes, if approved, have to be planned to fit within budget and schedule constraints. In this paper, we address the issue of assessing the difficulty of a change based on known or predictable data. This paper should be considered as a first step towards the construction of customized economic models for maintainers. In it, we propose a modeling approach, based on regular statistical techniques, that can be used in a variety of software maintenance environments. This approach can be easily automated, and is simple for people with limited statistical experience to use. Moreover, it deals effectively with the uncertainty usually associated with both model inputs and outputs. The modeling approach is validated on a data set provided by the NASA Goddard Space Flight Center which shows it has been effective in classifying changes with respect to the effort involved in implementing them. Other advantages of the approach are discussed along with additional steps to improve the results.

Key words: maintenance process, change difficulty, change request management.

[1] Research this study was supported in part by NASA grant NSG 5123 and the Vitro Corporation (IAP Member)

1 Introduction

Given the limited resources (i.e. effort and calendar time) available to the maintenance activity within software organizations and the number of change requests proposed, difficult decisions need to be made. These decisions include: which changes to implement, how much optional functionality to provide in enhancements. A large amount of total software effort is spent on maintenance [LS80, GRA87]. Changes in the form of corrections, enhancements or adaptations effect the software source code and/or the documentation. Some of these changes are crucial, others are less important. Therefore, when one considers the global cost and variety of maintenance activities, management of changes becomes an important and complex task. It requires the support of models so we may perform systematic comparison of the costs and benefits of changes before implementing them [RUV92]. One approach is to build such models based upon past project experiences.

To this end, effort models have to be designed to predict resource usage and optimize the cost-effectiveness of the maintenance process. Well defined modeling procedures need to be established so they can be repeated and refined, allowing the model to evolve consistently as new data are collected.

This paper describes a modeling procedure for constructing a predictive effort model for changes during the maintenance phase. This technique is intended to handle small data sets and the uncertainty (i.e. for cost or technical reasons) usually associated with model inputs and outputs (i.e. is this particular predication believable?). We assess the feasibility of building such a model using a data set that describes several projects in the SEL environment at the NASA Goddard Space Flight Center. Based upon the results of the analysis, we also make recommendations for improving the data collection process.

2 Context of Study and Experiment Design

In this study, we use a data set consisting of 163 changes collected on four different maintenance projects. Each change is represented by a vector consisting of a variety of metrics associated with the change. The four projects are referred to in the paper as projects p1, p2, p3, p4. These projects are from the same application domain: satellite ground support software written in FORTRAN.

The change process in the SEL environment has two main phases: an "understanding" phase where the change is determined and isolated in the system and an "implementation" phase, where the change is designed, implemented and tested.

The effort associated with both the understanding and implementation phases is collected on discrete scales (i.e. ordinal) in order to facilitate the data collection from a maintainer's perspective. The effort range is divided into five intervals: below one hour, between one hour and one day, between one day and one week, between one week and one month, above one month. For each change performed, the appropriate understanding effort and implementation effort intervals are recorded by the person making the change. These effort intervals are indexed from 1 to 5 and will be referred to as *difficulty indices* in the paper.

All the change-related data used in this paper was collected on a standard form (see Appendix). The metrics collected range from measures on a continuous scales (e.g., number of components added, number of lines of code added) to categorical measures (e.g., source of the change, technical description of the change). Some of these metrics are predictable before starting the design of the change, others can only be assessed after the implementation of the change has begun.

In this paper, we focus exclusively on the effort spent to implement (i.e design, code, test) a change. There are two reasons for this: 1) Almost no information is available to the

maintainer before the understanding phase. Therefore, no prediction model can be built. 2) In this environment, the effort expended in the understanding phase is generally somewhat smaller than the effort expended during the implementation. It is thus more essential to use a predictive model for the implementation phase.

The available metrics are defined as follows:

- Type of modification (correction, enhancement, adaptation).
- Origin of the error in the software life cycle. This is referred to as *source* in the text (requirements, specifications, design, code, previous change).
- Software products effected by the change (code only, code and design). This is referred to as *objects* in the text.
- Number of components added, changed, deleted. They are referred to as *comp.add*, *comp.ch.*, *comp.del.*, respectively.
- Number of lines of code added, changed, deleted. They are referred to as *loc. add.*, *loc. ch.*, *loc. del.*, respectively.
- Change technical description (initialization, logic/control structure, user interface, module interface, data structures, computational) . This metric is referred to as *ch.desc.*

During the *understanding phase*, estimates can be made of the first three metrics. The number of components involved in a change can also be approximated since the change is isolated in the system architecture. But any prediction in terms of lines of code to be added, deleted or changed is still complex at this point and can only be predicted at a coarse level of precision.

3 The Modeling Approach

Considering the discrete nature of the effort data reported during maintenance, the prediction issue becomes a classification issue, i.e. in which effort class will the change probably lie? The maintainer can only predict values for most input metrics with a certain degree of uncertainty. It is important that the modeling process takes this constraint into account. This help to make the generated model easy to use. Also, our data set is small and contains discrete explanatory variables. Therefore, we need a modeling approach which is both effective on small samples and which handles discrete and continuous explanatory variables in a consistent way.

3.1 The Modeling Process Steps

A high level view of the model construction process can be defined as follows:

1- *Identify Predictable Metrics*. Identify the metrics, among those available, that are predictable before the implementation phase. For ratio and interval metrics that are predictable early but only with a certain degree of uncertainty, the range is recoded as an ordinal range with a set of ordered classes. These classes reflect a reasonable level of prediction granularity. For example, a ratio level metric range like "number of components added" could be divided into three intervals forming the three metric classes low, average, and high.

2- *Identify Significant Predictable Metrics*. Identify a subset of the predicable metrics that appear to be good predictors of the difficulty index, using a consistent evaluation technique for all candidates.

3- *Generate a Classification Function.* Associate the resulting metrics in a classification function which has the following form:

Predicted_Difficulty = Classification_Function (Significant_Predictable_Metrics)

where Predicted_Difficulty = some classification scheme based on the difficulty indices, e.g., {easy, difficult} and Significant_Predictable_Metrics = {some of the predictable metrics collected on the Maintenance Change Report Form which appear as good predictors}

4- *Validate the Model.* Conduct an experiment on a representative (i.e. in terms of size and quality) set of data. Two measures that can be used to validate the model are: Average Classification Correctness (i.e. ratio of number of correct classification / total number of performed classifications), and Indecision Rate (i.e. ratio of number of undecidable cases / total number of changes to be classified). The latter reflects the need for such a model to deal with output uncertainty, therefore warning the user whenever a certain level of confidence is not reached for a specific prediction.

3.2 An Implementation of the Modeling Process

This section presents a possible implementation of the previously described process. Our goal in defining such a procedure can be described by the following points:

- We want the generated model to be as simple to use as possible.

- The uncertainty associated with the model inputs at the time of prediction must be taken into account by the model, i.e., intervals rather than values should be used as model inputs.

- The model should be able to provide some estimated risk of error associated with each classification. Thus, the user would be able to select a minimal level of confidence (i.e. maximum risk) that would differentiate the model classifications as believable or non-believable.

- The steps of the procedure are:

1- *Identify Predictable Metrics.* The input is a set of available metrics. The output is a set of metrics whose values are either known or predictable, with a certain degree of accuracy, before the change implementation phase.
There are several processes for selecting the set of predictable metrics. The determination of predictability can be either based on interviews with people with a good knowledge of the maintenance process (and then refined with experience) or observed through controlled experiments [BSP83, BW84]. Both help to determine the average estimation accuracy that can be reasonably expected for a given metrics.

The range of each continuous / ordinal predictable metric is divided into intervals (e.g., percentiles, natural clusters [DIL84]). The more accurately predictable the metric, the more numerous and narrow the intervals can be. We recode the metric ranges according to their respective predictability so the maintainer can easily select the right interval and use some of the predictive power of metrics not measurable before the *implementation phase*. These intervals are called *metric classes* in the paper.

Our need to define these metric classes for predictable metrics stems from the impossibility of relying exclusively on measurable (at the time of prediction) metrics, e.g. building an accurate model for predicting change effort is likely to require measures of change size that are not available before the implementation phase. We have no choice other than taking into consideration metrics that cannot be measured but only approximated with a certain degree

of precision by the maintainer after the *understanding phase* of the change process.

2- *Identify Significant Predictable Metrics.* The input to the second step is the set of predictable metrics from the first step and the outputs are a subset of significant predictors and their corresponding association table. This association table distributes the difficulty indices across the metric classes defined on each predictor value domain.

Consider as an example Table 1 which shows the association table of the metric *number of lines of code added* across the four difficulty classes (class 5 has so few changes that we merge it to class 4). This table is calculated based on the actual distributions in the data set considered for modeling. Each column represents a metric class (e.g. > 30 implies that the number of loc added is more than 30) and each row an index of difficulty. With respect to each predictable metric and using its calculated distribution of difficulty indices, an average difficulty index (i.e ADI) is calculated for each metric class (shown in the bottom row of Table 1). The calculation of a meaningful and statistically significant ADI requires us to set up the metric classes in a way that guarantees a minimum number of changes in each of them.

	Loc added		
DI	< 10	[10 30]	> 30
1	7%	0%	0%
2	48%	36%	9.5%
3	42%	60%	40.5%
4	3%	4%	50%
ADI	2.40	2.68	3.40

Table 1: "number of lines of code added" distribution

Taking the association table *Table 1* as an example, the calculated index averages look consistent with what was expected. The ADI seems to increase substantially with the number of lines of code added. In general, with respect to the ratio and interval level metrics whose the value domains have been recoded in successive metric classes (see step 1), significant differences should exist between class ADIs. Based on a F-test, a one-way analysis of variance (ANOVA) can be performed and the statistical level of significance of the metric class ADI differences may be estimated [CAP88]. Whenever the 0.05 level of significance is not reached, the boundaries should be recoded in a way that minimizes the level of significance. Since all the continuous metric ranges have been recoded into an ordinal scale, we have to calculate the degree of association between the difficulty indices and the metric classes in order to assess the predictive power of each metric. One approach consists of computing the Chi-Square statistic (which is valid at the nominal level [CAP88]) for each metric association table. A statistical level of significance characterizing the association between the difficulty indices and the metric classes is calculated based on the generated Chi-square value. Thus, the top ranked metrics showing sufficient degree of association are selected as parameters potentially usable to build a multivariate prediction model. Some more sophisticated measures of association (i.e. PRE-measures of associations [CAP88]) can provide more intuition/information about the associations and therefore allow an easier selection. However, this issue is beyond the scope of this paper.

3- *Generate a Classification Function.* The input to the third step is the set of association tables of significant predictable metrics and the output is a classification model that predicts an expected difficulty index associated with changes. Note that although five difficulty indices are defined on the change form, a small minority of the changes (5%) actually lie in the extreme intervals (i.e. intervals 1,5).

This makes classification into these intervals extremely difficult. Also, since 80% of the changes belong to classes 2 and 3, we will first build a classification model intended to differentiate these two classes: less than one day (i.e. referred as easy), more than one day (i.e. referred as difficult). In section 4.2, we will refine our classification by dealing with a "more than one week" class (i.e. indices 4 and 5). Thus, based on the generated classifications the user will be able to make decisions with respect to the requested implementations of changes. This is done by comparing the predicted difficulty to both the available resources and the expected gains.

The process of building a classification function is composed of two steps:

1- Perform a regression: Based on all the available association tables and the corresponding ADIs for each change in the data set, we perform a stepwise linear regression [DIL84] of the following form:

Actual_difficulty_index = W1 * ADI_metric1 + ... + WN * ADI_metricN

Due to interdependencies between metrics, only a subset of the preselected metrics remains in the generated prediction function (i.e. only the one showing, based on a F-partial test, a level of significance below 0.05). In order to make the model easier and less costly to use, the number of parameters in the regression equation can be minimized. In this case, one or several parameters are removed (especially when they show a statistical significance close to 0.05) and the resulting models are evaluated. Then, the user has to assess the loss of correlation against the ease of use gained by removing parameters from the model. If the tradeoff appears reasonable, then the new model is adopted. Weights are calculated for each remaining parameters and the resulting optimized linear function allows us to calculate an difficulty index expected value. This may be used to classify the change based on the realistic assumption that: the closer the expected value of the difficulty index to an actual difficulty index, the more likely the corresponding change belongs to the matching effort class. Therefore the following interval-based decision rule is used to make classifications.

2- Define a decision rule for classification: the predicted difficulty index range is divided into three intervals (i.e. easy change predicted, undecidable, difficult change predicted) in a way that guarantees a maximal average classification correctness. For example, the boundaries for classifying a change as either less or more than one work day can be defined as in Figure 1. The classification of future changes will be performed according to the interval in which their calculated difficulty index will lie.

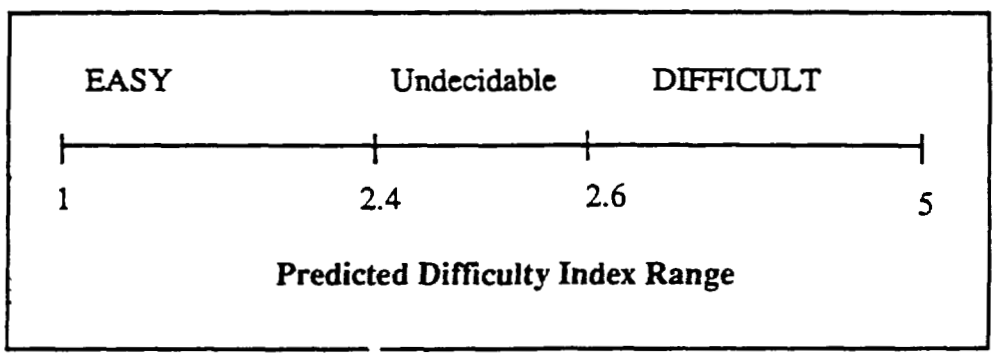


Figure 1 : Example of decision intervals

The process for creating these decision boundaries can be described as follows:

1- The user defines a *risk / loss function* having the following form:

$$\text{Expected\_loss} = \text{Weight1}*\text{MR1} + \text{Weight2}*\text{MR2}$$

where MRn is the misclassification rate calculated for changes actually in class n.

The loss function weights can be defined according to the respective costs of misclassification associated with each class. Most of the time, this weight will be set to one. A search algorithm can then be used to determine the interval between two neighboring changes on the predicted index range that provides the best decision boundaries (i.e. that minimizes the risk / loss function). These two neighboring changes form the boundaries of the smallest possible *undecidable interval* on the range.

2- In a stepwise manner, this interval can be widened on both sides of the index range according to some automatable process. For instance, the interval can be expanded in a stepwise manner, including one more change at a time on each side of the interval, until a maximal expected loss value (i.e. predefined by the user) is reached. Based on this process, the user will be able to determine the boundaries of the decidable intervals corresponding to the desired level of risk.

4 A Validation Study

According to the procedure defined above and based upon the previously described four project data set, the significance of each available metric as a predictor is assessed. Table 2 shows the Chi-square-based levels of Significance. Then, in order to build the needed classification models, the metrics yielding a good level of significance are selected. First, we build a general model usable for any project in the same environment. This model is intended to be useful at the start of a maintenance process when not enough data are available to create a project specific model.
Then, we build an independent classification model for each project which is expected to be more accurate with respect to future changes for each specific system, respectively. The various results will be compared in order to assess the validity of cross-project models in this environment. The ranges of the continuous metrics were recoded according to the previously described procedure. Two or three metric classes were defined for each of the metrics, according to the predictability level of the metric and the distribution of the changes on their respective range. In other words, the interval boundaries were chosen in a way that reflected their predicability, optimized the classification power of the metric (i.e. optimized the chi-square) and guaranteed, to the extent possible, a sufficient number of changes within each metric class.

Metric	Level of significance	Metric classes
Type	0.004	Correction, enhancement, adaptation
Source	0.0000	Requirements, specifications, design, code, previous change
Ch.desc.	0.0000	initialization, logic, interface, data structure, computational
Loc. add.	0.0000	<10, [10, 30] , >30
Loc. Ch.	0.0000	<10, [10, 25] , >25
Loc. Del.	0.0006	<2, [2, 15] , >15
Comp. add.	0.0002	0, > 0
Comp. ch.	0.0000	<2, [2, 5] , >5
Objects	0.0005	code only, code and design

Table 2: Level of significance and class boundaries / categories of metrics

4.1 A General Model

This model is intended to be specific to the NASA SEL environment. It has been built based on systems belonging to the same application domain and therefore may not represent necessarily other domains accurately. Table 3 shows for each selected metric, the class ADIs and the corresponding result of the one way analysis of variance [CAP88] that assessed the statistical significance of the ADI variations across metric classes. They all appear below 0.05 and we can therefore say that the metric classes with respect to continuous metrics have been adequately defined because they show significant ADI differences.

Table 4 shows two distinct regression-based classification functions (PI stands for Predicted difficulty Index). Note that the parameters of the regression equations are the metric association table-based ADIs and not the metric values themselves. For the sake of simplification, the names of the metrics are shown in the equations. For each function, the calculated regression equations are given with the respective level of significance of each metric (i.e.shown between brackets above the equations and based on partial F-tests).

If the metric does not appear significant at a 0.05 level, then they are excluded of the equation. The global coefficient of determination R^2 is also given. The first one was obtained by performing a stepwise regression using the class ADIs of the significant predictable metrics. Only one of the lines of code (i.e. loc) based metrics was retained in the equation: *loc.ch*. Then, in an attempt to avoid the use of this metric (i.e. which is still the most difficult to assess despite the coarse defined metric classes), we recalculated the equation parameters when ignoring it. The coefficient of correlation did not appear much affected by the change. This can be explained by the higher significance of the remaining parameters and their stronger calculated coefficients that show a strong interdependence with *loc.ch*. In other words, they partially compensated the loss of explanatory power due to the removal of *loc.ch*.. Thus, the generated model becomes even easier to use and does not loose much of its accuracy (see Table 5).

Metrics	Level of significance	ADIs for each category
Type	0.003	[2.56 , 2.36, 2.95]
Source	0.0000	[3.04, 3.29 2.42, 2.33, 2.27]
Ch.desc.	0.0000	[2.2, 2.9, 3.0, 3.1, 2.4, 2.9, 2.8]
Loc. add.	0.0000	[2.4, 2.68, 3.4]
Loc. Ch.	0.0000	[2.4, 2.8, 3.14]
Loc. Del.	0.0001	[2.6, 2.8, 3.4]
Comp. add.	0.0000	[2.64, 3.63]
Comp. ch.	0.0000	[2.41, 3.0, 3.31]
Objects	0.01	[2.63, 3.03]

Table 3: Metric class ADIs

	Description of the Models	R-sq
Model 1	(0.0) (0.0) (0.0) (0.0008) (0.04) (0.004) PI = - 4.22 + 0.59 Source + 0.62 Ch.desc + 0.58 loc.ch + 0.38 Comp.add + 0.36 Comp.ch	0.50
Model 2	(0.0) (0.0) (0.0) (0.01) (0.0) PI = - 3.95 + 0.68 Source + 0.69 Ch.desc + 0.49 Comp.add + 0.56 Comp.ch	0.46

Table 4: General models

Table 5 shows the *classification correctness* (i.e. rate of correct classification) obtained when using the above models (Table 4). The decision boundaries have been optimized to yield the best results. First, they have been selected to yield a 0% indecision rate (column IR = 0% in Table 5). Then the undecidable interval has been widened in order to demonstrate the possibility of selecting decision intervals that fit the user's need in terms of classification correctness (column IR > 0% in Table 5). In this case, the selected interval boundaries are arbitrary and are shown for the sake of example. The row "classification" indicates the classification performed (i.e. easy changes = [1-2] or [1-3]). Each cell contains, for all models, the undecidable interval boundaries between brackets and the corresponding classification correctness. Whenever the undecidable interval has been widened (i.e. IR > 0%), the corresponding indecision rate is given.

Despite the mediocre coefficient of determination, a particularly good correctness has been obtained when the interval [1-3] represents easy changes. However, the results appear much less satisfactory for the other classification performed. Nonetheless, this can be substantially improved by widening the undecidable interval. Thus, the model appears usable for at least a subset of the changes. However, when possible (i.e. enough project data are available), project specific models should be used as demonstrated in the next

paragraphs.

4.2 Project Specific Models

Table 6 shows optimal equations resulting from stepwise regressions performed independently for each of the four projects. The format used is the same as in Table 5. Differences between models are observable with respect to the variables selected. This does not necessarily mean a real variation in the impact of the explanatory variables across projects. It may be due to a lack of variation of a variable within a project specific data set.

	Description of the Models	R-sq
Model P1	(0.03) (0.0023) (0.0002) PI = - 1.56 + 0.71 Source + 0.80 Comp.ch	0.68
Model P2	(0.0) (0.004) (0.003) PI = - 3.62 + 0.65 Source + 0.80 Ch.desc	0.45
Model P3	(0.001) (0.0001) (0.0016) (0.009) PI = - 1.34 + 0.59 Ch.desc + 0.50 Loc.add + 0.44 Comp.ch	0.75
Model P4	(0.002) (0.003) (0.001) PI = - 2.95 + 0.65 Loc.add + 1.02 Loc.ch	0.50

Table 6: Project specific regression equations

The correctness is shown to improve substantially (see Table 7), compared to the general model results whenever easy changes = [1-2] (except for project P2). The results are only presented for a minimal undecidable interval. However, the interval could be widened as shown in the previous section in order to get even better correctness in the decidable intervals.

PROJECT MODEL RESULTS		
Indecision	IR = 0%	
Classification	clas. [1-2] / [3-5]	clas. [1-3] / [4-5]
Model P1	[2.39 2.80] : 88%	[3.35 3.74] : 88%
Model P2	[2.31 2.52] : 74%	[3.42 3.61] : 93%
Model P3	[2.45 2.54] : 87%	[3.47 3.55] : 92%
Model P4	[2.45 2.54] : 81%	[3.47 3.55] : 89%

Table 7: Classification results

5 Conclusions, Lessons Learned and Future Research

This modeling approach provides a simple and flexible way of classifying changes during the maintenance process. The classification power of continuous explanatory variables can be optimized by changing the class boundaries until the chi-square statistic reaches a maximum (this can be automated). This is performed while minimizing the number of metric classes and thereby facilitating the prediction process. It allows for an optimal use of the available explanatory variables by considering the uncertainty associated with each of them at the time of prediction.

A user defined loss function (i.e. risk model) can be minimized while selecting the decision boundaries on the predicted index range until a predefined expected loss is reached. This allows the construction of a classification model optimal and customized, for specific user needs. Thus, by tuning the undecidable interval, he / she can handle in an appropriate and simple way the uncertainty associated with the model output. Also, the modeling process has shown many opportunities for a high extent of automation that would help optimize the metric class definitions and select the most suitable decision boundaries.

Despite the fact that collecting change effort data on a discrete range (i.e. ordinal level) makes the data analysis more difficult and the usable statistical techniques less powerful, valuable information can still be extracted from the data while taking into account the constraints associated with a software development environment. As presented, effective classification has been performed among three effort classes with respect to changes within the maintenance process.

Despite organizational issues and data collection accuracy problems, it would be better to collect effort data at a ratio level. This would allow the use of more effective statistical techniques. The gains in terms of management efficiency are likely to be substantial. However, if effort data are collected in a discrete manner, each class should contain, to the extent possible, the same number of changes. When the distribution is not uniform, classification for small proportion classes may be difficult.

Sub-system and component characteristics that are collectible in an automated way through code static analyzers (i.e. data binding between components, code complexity, ...) are likely to help refine the classification models. Maintainer skills and experience with respect to the maintained system should also be considered in the analysis in order to better select the required level experience for minimizing the cost of maintenance. Despite encouraging average results in the above experiments, a more complete data collection process is required in order to refine these change difficulty prediction models.

6 Acknowledgements

We would like to thank Jon Valett from the NASA Goddard Space Flight Center, Adam Porter and Chris Hetmanski for their suggestions that helped improve both the content and the form of this paper.

7 References

[BSP83] V. Basili, R. Selby and T. Phillips. "Metric Analysis and Data Validation across FORTRAN Projects". IEEE Transactions on Software Engineering, SE-9(6):652-663, November 1983

[BW84]V. Basili and D. Weiss. "A Methodology for Collecting Valid Software Engineering Data". IEEE Transactions on Software Engineering, SE-10(6):728-738,

[CAP88] J. Capon, "Statistics for the Social Sciences", Wadworth publishing company, 1988.

[DIL84] W. Dillon and M. Goldstein, "Multivariate Analysis", John Wiley & sons, 1984.

[GRA87] R. Grady, "Software Metrics: Establishing a Company-Wide Program", Prentice-hall, 1987.

[LS80] B. Lientz and E. Swanson, "Software maintenance management", Addison-Wesley, 1980.

[RUV92] D. Rombach, B. Ulery and J. Valett, "Toward Full Cycle Control: Adding Maintenance Measurement to the SEL", Journal of systems and software, May 1992.

SECTION 5 – SOFTWARE MEASUREMENT

SECTION 5—SOFTWARE MEASUREMENT

The technical paper included in this section was originally prepared as indicated below.

- "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," H. D. Rombach, B. T. Ulery, and J. D. Valett, *Journal of Systems and Software,* May 1992

Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL

H. Dieter Rombach and Bradford T. Ulery
Computer Science Department and Umiacs, University of Maryland, College Park, Maryland

Jon D. Valett
NASA, Goddard Space Flight Center, Greenbelt, Maryland

Organization-wide measurement of software products and processes is needed to establish full life cycle control over software products. The Software Engineering Laboratory (SEL)—a joint venture between NASA's Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation—started measurement of software development more than 15 years ago. Recently, the measurement of maintenance has been added to the scope of the SEL. In this article, the maintenance measurement program is presented as an addition to the already existing and well-established SEL development measurement program and evaluated in terms of its immediate benefits and long-term improvement potential. Immediate benefits of this program for the SEL include an increased understanding of the maintenance domain, the differences and commonalities between development and maintenance, and the cause-effect relationships between development and maintenance. Initial results from a sample maintenance study are presented to substantiate these benefits. The long-term potential of this program includes the use of maintenance baselines to better plan and manage future projects and to improve development and maintenance practices for future projects wherever warranted.

1. INTRODUCTION

Most software organizations lack satisfactory control over their development and maintenance projects. This lack of control is exemplified by the absence of explicit models enabling the identification of ambiguous product requirements, the selection of practices best suited to achieve given requirements, or the prediction of the impact early project decisions may have on the quality of the resulting products. Each organization has its own set of control problems and reasons standing in the way of improvement. Comprehensive measurement programs are needed as a first step toward improvement [1]. Such programs can help identify the specific problems of an organization in quantitative terms, pinpoint possible causes, motivate improvements, and assess alternatives considered for improvement.

The Software Engineering Laboratory (SEL)—a joint venture including government, industry, and university— began measurement of satellite ground support software development projects in 1976. The three primary organizational members of the SEL are the Systems Development Branch at NASA's Goddard Space Flight Center, the Computer Science Department at the University of Maryland, and the Systems Development Operation at Computer Sciences Corporation. This collaboration has produced numerous case studies and controlled experiments [2-6]. Results from these case studies and experiments motivated several improvements within the SEL [7-9].

In 1988, the SEL incorporated maintenance into its scope of measurement. The result is an even more comprehensive measurement program in which data is now being collected during development and maintenance of all software systems. In the SEL, pre- and postlaunch maintenance activities are performed by separate organizational entities. Currently, maintenance data are only collected from prelaunch maintenance activities. In the remainder of this article, the term "maintenance" shall refer to this prelaunch phase

Address correspondence to Professor H. Dieter Rombach, AG S‡ W Eng., Fachbereich Informatik, Universitaet Kaiserslautern, Postfach 3049, D-6750 Kaiserslautern, Germany.

between delivery of a completed software system and the actual launch of the related spacecraft. This maintenance measurement program is customized to the pertinent SEL characteristics, including the definition of maintenance, the maintenance improvement goals, and other product, process, and people factors.

Empirical research in the SEL is based on the idea of continuous improvement. This idea has been formulated as the quality improvement paradigm [1]. According to this paradigm, improvement is the result of continuously understanding current practices, changing them, and empirically validating the impact of these changes. Improvement requires measurement.

In the SEL, measurement goals define the data to be collected and provide the context for data interpretation. This goal-oriented approach to measurement has been formulated as the goal/question/metric paradigm [1, 10, 11]. It suggests defining each goal by developing a set of analysis questions, which in turn lead to a set of metrics and data. The short-term goals of our maintenance measurement program have been to increase the understanding of maintenance within the SEL; the long-term goals are to stimulate improvements in the SEL's ability to plan and manage future maintenance projects and—whenever needed—to motivate the use of different development and maintenance practices.

Specific characteristics of the SEL maintenance environment as well as the comprehensive scope of our measurement approach make this program unique. The study results presented here may not be directly comparable to those from other maintenance environments, yet they do show how a comprehensive measure program can be used to better understand and improve an organization's development and maintenance process and products. Few comprehensive maintenance studies have been published [12-14]. Most empirical maintenance studies report on laboratory-style controlled experiments [15, 16], isolated case studies [13, 17], or project surveys [18]. A survey of maintenance studies has been published by Hale and Haworth [19].

The purpose of this article is to state our initial maintenance study goals and questions, present the related results, and propose—based on what we have learned—a revised set of goals and questions for future studies.

The study results are organized according to the types of data used to address the goals and questions: quantitative maintenance baselines, comparisons between quantitative development and maintenance baselines, and qualitative information regarding the cause-effect relationships between development and maintenance. These results have increased our understanding of maintenance processes and maintained products in the SEL, commonalities and differences between development and maintenance, and development characteristics affecting maintenance. On occasion, our results are carefully compared with results from other published studies or widely believed maintenance myths.

We begin our presentation with a background discussion of the SEL and the new maintenance measurement program (sections 2 and 3, respectively). We then present the results of our study (section 4). We conclude with an assessment of the SEL maintenance measurement program and a revised set of goals and questions for future maintenance studies.

2. THE SEL

The goals of the SEL are to understand its software development processes, to measure the effects of various methods and tools on these processes, and to identify and then apply new, improved development practices. Improved understanding within this particular environment provides the basis for better planning and management as well as a rationale for adopting new practices [4].

Development in the SEL supports satellite missions. SEL studies generally focus on attitude ground support systems and their associated simulators. These product lines are very stable: the system architecture, documentation standards, and organizational responsibilities do not change significantly from one mission to another. Attitude ground support systems have 130-240K lines of FORTRAN source code (where a line of code is measured as a physical line, including comment lines) and require 15-30 staff years to develop. Simulators have 25-75K lines and require 3-10 staff years to develop.

Research in this environment is guided by two basic paradigms: the quality improvement paradigm (QIP) and the goal/question/metric paradigm (GQM). The QIP, which applies the principle of continuous improvement to software engineering, defines the context for measurement within the SEL [1]. Accordingly, software development can be improved by iterating the following steps for each project: (1) characterize the corporate environment; (2) state improvement goals in quantitative terms; (3) plan the appropriate development practices and methodologies together with measurement procedures for the project at hand; (4) perform the development and measure, analyze, and provide feedback; and (5) perform postmortem analysis and provide recommendations for future projects. Each QIP iteration is characterized by its own set of goals. These

goals reflect—and evolve with—the maturity of the investigated organization.

Measurement in the SEL is guided by the GQM paradigm [10]. Measurement is used to characterize current development practices, monitor and manage development projects, identify strengths and weaknesses of the current practices, and evaluate promising new technologies in a controlled environment. The GQM paradigm describes a goal-oriented approach to measurement in which metrics are tied to specific measurement goals. According to the GQM paradigm, each measurement goal is listed explicitly, a set of specific questions is posed to address each goal, and specific metrics and measurement procedures are defined to support the questions. The resulting data collection procedures and interpretations are tailormade to the study's goals and local environment characteristics. For instance, in the SEL, this generally means that metrics and measurement procedures reflect the use of SEL-specific development practices, fit the organizational structure, and permit comparisons with historical data. Goals, questions, and metrics provide a context that helps ensure that data are interpreted correctly and are compared only to data and results from similar contexts.

Two types of measurement are common in the SEL: routine monitoring and exploratory studies. Routine monitoring is used to characterize the local environment broadly. The resulting quantitative and qualitative baselines are used to plan and manage new projects and to compare the effects of newly introduced tools or methods against [6]. Objective and subjective data are routinely gathered for each project [20]. Objective data include staff hours, computer utilization, source code growth, and the number and kinds of changes made to the source code. Subjective data characterize the software development process and software product characteristics. The data for over 100 projects monitored over the last 15 years is maintained in the SEL database [21].

Exploratory studies are used when the SEL is in the initial phase of understanding a process or methodology. For example, the SEL is currently studying three projects following the cleanroom methodology [22]. Special data collection procedures were designed for these projects to permit researchers to monitor the effort spent in reading and reviewing designs and code.

Measurement in the SEL has provided a rationale for making evolutionary changes to NASA's development practices, including stricter use of code-reading techniques [5], guidelines for Ada projects [23], and the adoption of the cleanroom development approach [24]. With the addition of maintenance measurement, the SEL is attempting to lay the foundation for similar improvements in maintenance.

3. THE SEL MAINTENANCE MEASUREMENT PROGRAM

The following subsections describe the SEL maintenance environment and the specific goals and procedures of our measurement program. A more detailed description of this environment, its products, and maintenance processes appeared in the proceedings of the 1989 IEEE Conference on Software Maintenance [25].

3.1 Maintenance Environment

In the SEL, maintenance is partly defined by organizational responsibility and schedule. As depicted in Figure 1, each product passes through three different organizational units during its lifetime: analysts produce the initial functional specifications used by the developers and remain responsible for these specifications throughout development and until launch; operations assumes complete responsibility after launch. During the period between development and launch, the analysts have complete responsibility for the system, including the implementation of any changes.

In this study, maintenance refers specifically to software change activities performed by the analysts during the postdevelopment, prelaunch phase. By nature of these constraints, the maintenance phase is typically shorter in the SEL than in other environments (one to two years), and the maintenance changes are not triggered by operational failures but by failures detected during simulated uses of the software by prospective operators and externally triggered changes of the overall satellite mission.

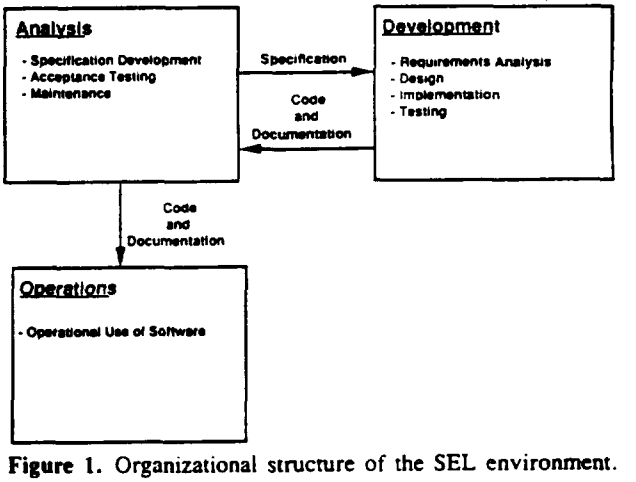


Figure 1. Organizational structure of the SEL environment.

The products maintained are the same simulators and attitude ground support systems described in section 2. Typically, the effort expended during the one- to two-year time frame that these systems are in maintenance is approximately 5% of the development effort. Maintenance procedures vary from project to project depending on the type of system being maintained, the size of the maintenance team (2–10 people on the projects studied), the specific methods and tools elected by the individual programmers, and other factors. In general, formal change control procedures are followed; changes are implemented one at a time, but may be tested in groups; and one maintainer is responsible for implementing each change.

3.2 Maintenance Measurement Goals

Consistent with the overall directions of the SEL, we chose three general goals for the maintenance measurement program: (1) to understand maintenance processes and products better; (2) to improve our ability to manage current maintenance projects and plan future ones; and (3) to establish a sound basis for improving development from a maintenance perspective.

Following the QIP, the initial goals focus on understanding maintenance. Representative measurement goals and questions selected for this study are summarized in Figures 2, 5, and 11. Analysis results related to these goals and questions are presented in section 4.

3.3 Maintenance Measurement Procedures

The data collection procedures used in this study were designed according to the principles of the GQM paradigm. Data were collected via exploratory interviews and routine data collection forms [20]. The routine data collection forms used during maintenance include the Weekly Maintenance Effort Form and the Maintenance Change Report Form (Appendix A). The effort form is filled out once per week per maintainer per system; one change form is filled out per completed change. The weekly effort forms record the distribution of effort (in staff hours) by type of change (correction, enhancement, adaptation, or other[1]) and by engineering activity (designing, coding, etc.). The change forms record the distribution of changes by type of change, size of change, changed objects (e.g., code, user's guide), expended staff time, fault type (if applicable), and more. All data are validated through a series of checks by the data entry personnel, project managers, and SEL researchers. Data are stored and made available to researchers and developers through the SEL database [21].

[1]All maintenance effort that cannot be attributed to an individual maintenance change is classified as "other." This includes effort related to management, meetings, and training.

4. MAINTENANCE MEASUREMENT BENEFITS

The maintenance measurement program has already increased understanding of maintenance in the SEL. Previously, much of this understanding was at best intuitive and approximate. In this section we demonstrate what we have learned as a result of our initial study. The results are separated into baseline characterizations of maintenance, a comparative analysis of development and maintenance, and an analysis of how development decisions affect maintenance.

In this study, we restrict our analyses to three large attitude ground support systems for which we have complete and valid data: the Gamma Ray Observatory, the Geostationary Operational Environmental Satellite, and the Cosmic Background Explorer. Maintenance of these systems was performed between 1988 and 1991. A total of 90 changes and over 10,000 hours of effort serve as the basis for all quantitative analyses of maintenance presented here.

Examining the data on these three projects has provided valuable insight into the maintenance process within this environment. The results presented here are intended to demonstrate the increased understanding of the maintenance process that can result from a measurement program.

4.1 Maintenance Baselines

The first step toward understanding any environment is to develop baselines describing that environment [12, 14]. The goals and questions related to this part of the SEL study are listed in Figure 2. They are intended

GOAL 1: Characterize the changes performed during maintenance.
  QUESTION 1
    How many changes of each type are completed?
  QUESTION 2
    How much effort is spent on changes of each type?
GOAL 2: Characterize product evolution during maintenance.
  QUESTION 3
    How much code is affected by each change?
  QUESTION 4
    Is code added, changed or deleted?
GOAL 3: Characterize the maintenance process stability
  QUESTION 5
    How do maintenance processes differ across projects?

Figure 2. Measurement goals for understanding maintenance in the SEL.

to characterize what kinds of changes are performed during maintenance, which parts of the systems change and how, and what maintenance processes are followed. In the long term, the resulting baselines are expected to provide a basis for determining whether new techniques or process adjustments have any measurable impact on the SEL maintenance processes or products. Any comparison between SEL baselines and baselines from other environments must take environmental differences into account.

Each maintenance change in this environment is well defined by a formal change request. There are several key steps in the change process: changes must be approved, implemented, tested, and released. In general, more changes are approved than can be implemented. This poses the difficult management problem of selecting which changes to implement. This decision is based on the importance of the changes approved as well as the budget available to make changes. The implementation of a change is performed by one programmer; there is no standard, formal methodology. Testing, beyond debugging by the programmer, is performed for several changes at once. One important implication is that the associated effort measured cannot be ascribed to a particular change. In fact, testing is typically performed at two levels: the first level provides internal checkpoints for configuration management; the second level occurs before each release.

Each maintenance change performed in the SEL is classified as an enhancement, adaptation, or correction [26]. A simple count of changes suggests that maintenance is primarily corrective; however, the effort distribution reveals that most effort is actually related to enhancements (Figure 3). Either way, adaptations do not seem to contribute significantly (Figure 2, questions 1 and 2). Note that the average enhancement requires just over twice the effort of the average correction. This phenomenon could be caused by the fact that enhancements are typically larger than corrections, that enhancements are inherently more difficult to accommodate into an existing system, or both.

As early as 1976, Belady and Lehman [14] demonstrated the benefits of program evolution models for the purpose of understanding the decay of software undergoing change. Figure 4 summarizes how many modules and lines of source code have been added, changed, or deleted per change (Figure 2, questions 3 and 4). On average, three lines of code are added for every existing line changed or deleted. Entire modules are rarely added and never deleted. In the SEL, maintainers do not significantly alter the system's architecture to make changes. We hypothesize that the high number of lines added reflects the high proportion of enhancements, and that architectural stability reflects an "if it ain't broke don't fix it" attitude. Such an attitude could be explained by the general lack of understanding of overall system architecture. The observed growth pattern also suggests that module functionality increases during maintenance, leading to a decrease in module cohesion. Decreased cohesion may not be a problem during the short lifespan of a satellite system, but may reduce the reuse potential of modules in future developments.

Our most striking observation about SEL maintenance is the extent to which the maintenance processes vary across similar projects (Figure 2, question 5). Some of the variability reflects the size and composition of the maintenance teams (2–10 programmers). One particular area where the processes differ appears to be in the approach to testing. The projects studied have not established well-defined criteria for when system or integration testing should be performed during maintenance. Such variability in the process reflects the relatively ad hoc nature of the maintenance environment as compared to the development environment. In fact,

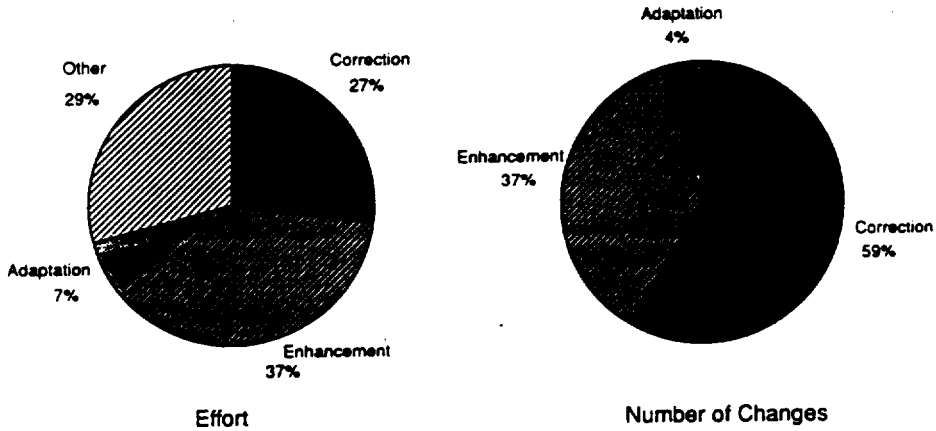


Figure 3. Distributions of effort and number of changes by type.

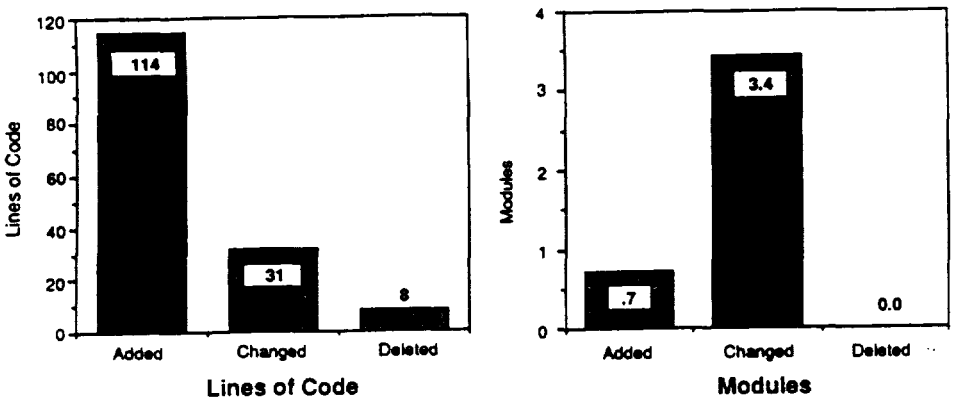


Figure 4. Lines of code and modules per change.

studies such as this one aim at increasing the maturity of the maintenance process within this environment. By identifying which aspects of the process are most successful, a single consistent process will be identified.

4.2 Maintenance vs. Development

Applying experience from past development studies to maintenance requires an understanding of the similarities and differences between maintenance and development. The goals of this part of the study were to compare changes made during development and maintenance, types of changes, and change processes (Figure 5). These comparisons are possible because both development and maintenance data are available for the three systems studied.

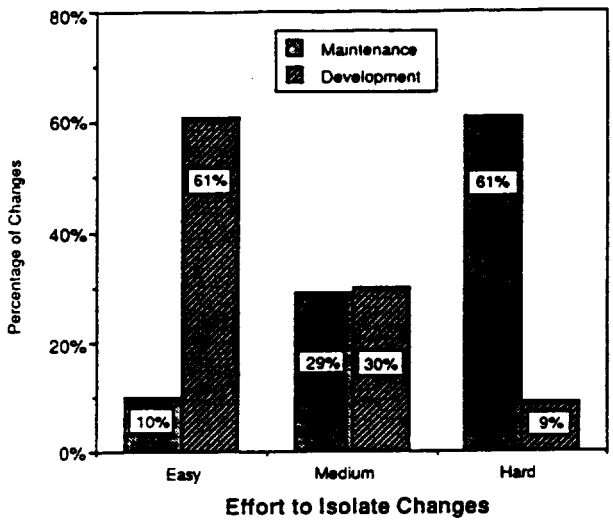
Throughout development and maintenance, the effort spent on each change is recorded. Effort is classified as easy when it takes less than an hour to complete a change, medium when it takes between an hour and a day, and hard otherwise. A distinction is made between the effort to isolate a change (understand the request and locate the affected modules) and the effort to complete the change (design, code, test). Figure 6 shows that changes performed during maintenance generally require more effort than those performed during development (Figure 5, question 6). We consider two hypotheses that might account for this pattern: changes requested during maintenance are inherently harder than those requested during development; and it is more difficult to perform the same change during maintenance than it would be during development. While we cannot determine whether particular modules are easy or difficult to change during maintenance based on our data, we are able to examine both hypotheses further at the level of the individual change.

GOAL 4: Compare changes made during development and maintenance.
  QUESTION 6
    How does the effort per change compare?
GOAL 5: Compare the types of changes made to products at both phases.
  QUESTION 7
    Are the faults found during maintenance different than those found during development?
  QUESTION 8
    How do the distributions of errors by class compare?
GOAL 6: Compare change processes at both phases.
  QUESTION 9
    How does the distribution of effort by activity type compare?

Figure 5. Measurement goals for understanding the similarities and differences between development and maintenance.

Regarding the first hypothesis, we find no obvious difference between the effort distribution patterns for all changes (Figure 6) and corrections only (Figure 7). We conclude that the increased effort is not primarily due to differences in the distributions of types of changes requested.

Regarding the second hypothesis, various characteristic differences between development and maintenance are commonly thought to explain why the same change might be more difficult to perform during maintenance. These include product factors (such as increased complexity and missing or out-of-date documentation), process factors (such as schedule constraints, methods, and tools), and people factors (such as a lack of familiarity with the software). In the SEL, we cannot attribute the maintenance difficulties to product factors because there is already a sharp increase in change effort during acceptance test, but little change in the products. Instead, we suspect some combination of process and people factors. Although we are unaware of any significant methodological differences between the



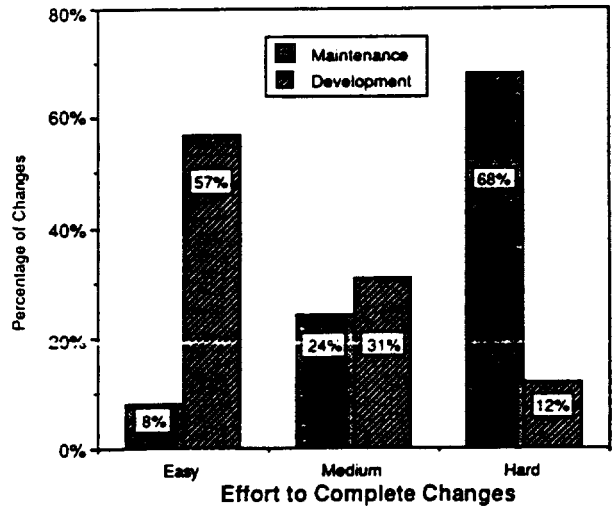
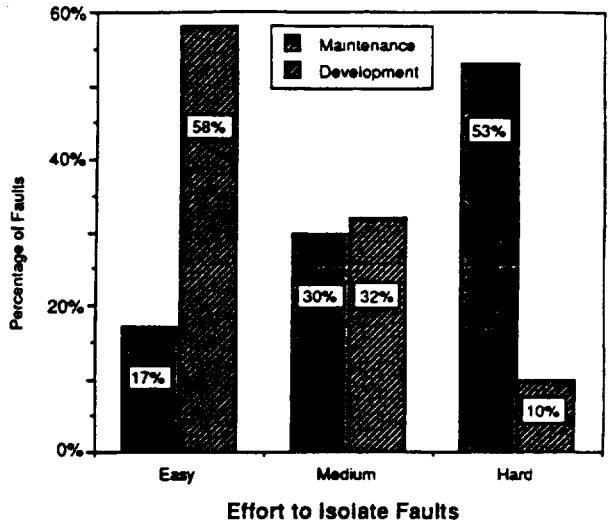


Figure 6. Effort to isolate and complete changes: maintenance vs. development. Easy, ⟨1 hour; medium, ⟩1 hour and ⟨1 day; hard, ⟩1 day.



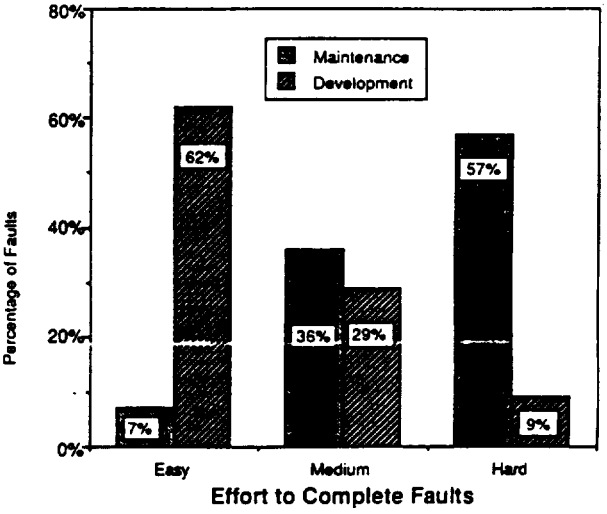


Figure 7. Effort to isolate and complete faults: maintenance vs. development. Easy, ⟨1 hour; medium, ⟩1 hour and ⟨1 day; hard, ⟩1 day.

way a change is implemented during development or maintenance, development has a much higher rate of change activity: these systems average over 1,000 changes during testing. Although the high number of changes may increase certain costs (e.g., configuration control), it may actually reduce others (e.g., testing is not repeated once for every change). Maintainers are not only generally unfamiliar with the systems they maintain, but the volume of maintenance may be insufficient to develop such familiarity. We expected the unfamiliarity with the maintained systems to have a more dramatic impact on the isolation activity (which might require an understanding of the entire system) than the completion activity (which typically requires only an understanding of individual modules). Instead, we discovered a proportional increase in both isolation and completion efforts (Figure 6). This may be explained by the fact that SEL maintainers are experts in the application domain, not software development; therefore, they may be expected to readily understand the change specifications, but not the code.

Both during development and maintenance a significant fraction of the changes are corrections (Figure 3). Figure 8 shows that the types of faults corrected during development and maintenance are similarly distributed (Figure 5, question 7). During maintenance, more corrections are related to incorrect initialization (21 vs. 17%) and logic (25 vs. 19%), but fewer are related to incorrect interface (19 vs. 22%), data (26 vs. 28%), and computation (9 vs. 14%) as compared to develop-

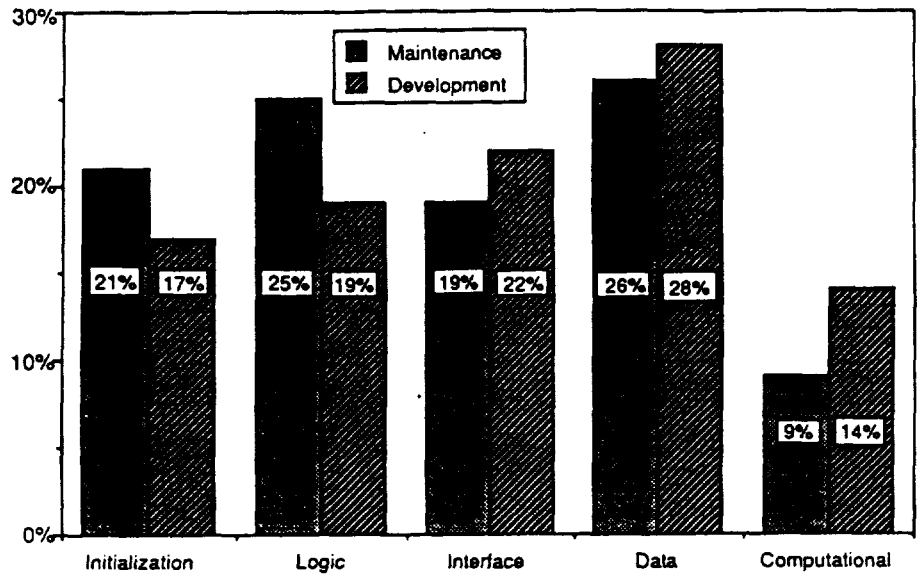


Figure 8. Number of faults per class: maintenance vs. development.

ment (see [20] for definition of classification scheme). Some of the differences seem to be related to the organizational structure of the environment. Maintenance is performed by people more familiar with the application domain and less familiar with the solution domain. The opposite is true for the developers. In this environment, many application-specific parameters are reflected in the software as initialization parameters. As such, they require a clear understanding of the application, and faults are more easily found by maintainers. The opposite is true for typical solution faults such as interface and computational faults.

Figure 9 shows that the distributions of errors differ significantly between maintenance and development (Figure 5, question 8). During maintenance, many more faults are attributed to inappropriate requirements or specifications (26 vs. 3%), and a few more are attributed to inappropriate design (11 vs. 8%); fewer are attributed to inappropriate implementation (55 vs. 79%) or previous changes (2 vs. 10%). In attempting to explain these differences, the following hypotheses have been formulated. Few faults are attributed to previous changes during maintenance because maintainers are unaware of changes made during development and the

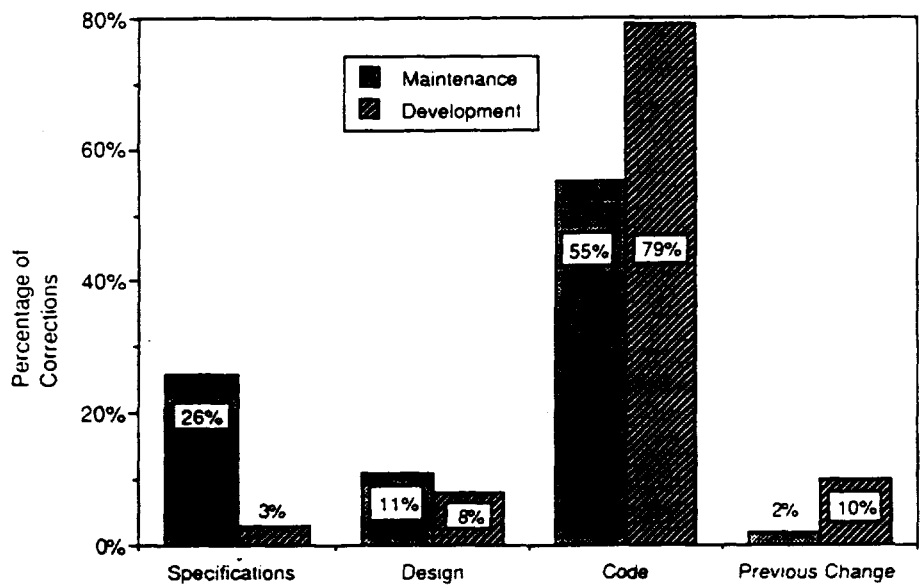


Figure 89. Number of faults per source: maintenance vs. development.

total number of changes during this phase is low. The high proportion of faults attributed to the requirements or specifications reflects the nature of the testing: applications experts are now using the systems to prepare for the missions, whereas during development most testing is performed by the developers themselves.

During development and maintenance, effort data is collected according to the following process model: isolation (understanding a requested change and identifying the affected modules), design (proposing a change), implementation (implement the proposed change), unit and system test (testing the changed modules and system), and acceptance test (testing a set of related changes). The development data include all effort; it is not limited to changes.

Figure 10 shows that during maintenance, more effort is spent on design activities, about the same amount of effort is spent on implementation activities, and less effort is spent on testing activities (Figure 5, question 9). The increase in design effort may be explained by a lack of familiarity with the system structure, resulting in increased effort to isolate changes. The decrease in testing effort may be explained by different testing procedures. During maintenance, integration testing is almost absent because the system structure doesn't change much, and acceptance testing is performed for groups of changes together.

How do these results compare with similar findings published in the literature? While comparing baseline data across environments is difficult, some patterns are evident. The increased cost of maintenance changes and corrections has been noted previously by many authors [22, 27]. This lends support to the claim that faults introduced during design but discovered during maintenance may cost significantly more than if discovered and corrected earlier in the life cycle [27]. As has been noted in other environments [28], we find that maintenance changes in the SEL require more "up-stream" (i.e., design) than "down-stream" (i.e., testing) effort).

4.3 Development for Maintenance

As a final result of the maintenance measurement program, the SEL has enhanced its understanding of the impact of development decisions on maintenance (Figure 11). This increased understanding is illustrated by our initial findings concerning the complexity of delivered products and the quality of their documentation. The qualitative results of this section are based primarily on subjective data from exploratory interviews. Nevertheless, they are essential during the early phases of a measurement program for guiding future improvement cycles.

Our initial inquiries have revealed complexity problems related to intermodule structure and the encoding global information (Figure 11, question 10). Maintainers reported major problems related to the fact that global information was encoded redundantly. For example, constants were encoded in multiple FORTRAN common blocks. Software modification frequently resulted in inconsistent representations of global information.

Two recurrent documentation problems have been identified (Figure 11, question 11). These concern the

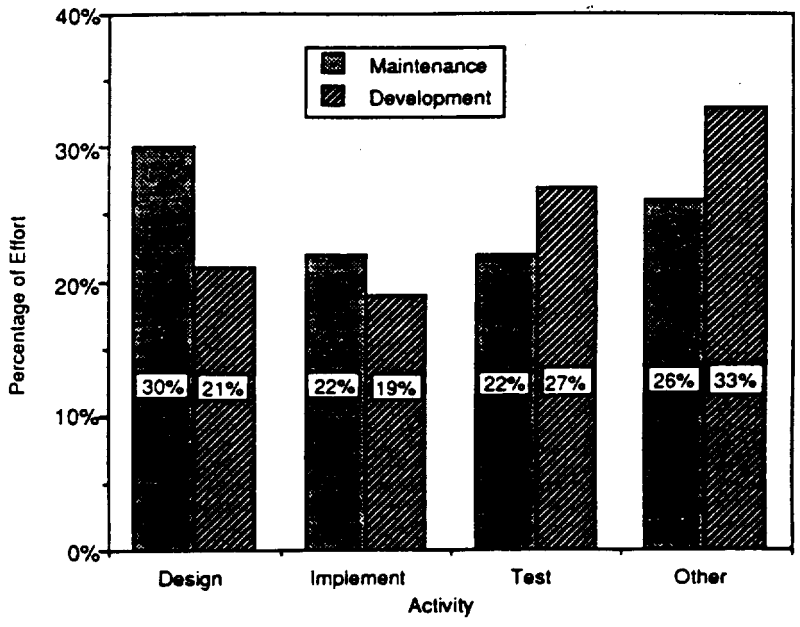


Figure 10. Effort per activity: maintenance vs. development.

GOAL 7: Characterize the impact of the delivered product on maintenance.
QUESTION 10
What structural product characteristics have positive/negative effects on maintenance?
QUESTION 11
What product documentation standards have positive/negative effects on maintenance?

Figure 11. Measurement goal for understanding the effects of development on maintenance.

use of program design language (PDL) and debug statements. PDL descriptions of each module are included in the source code as a header. Most maintainers regard PDL as redundant. Furthermore, the delivered PDL is usually outdated. In the SEL environment, developers are required to keep their design PDL as part of the software module. Unfortunately, this PDL is frequently obsolete by the time the module reaches the maintenance phase; thus, it is useless to the maintainers. Also, the majority of people maintaining the software suggested that this practice be stopped entirely, since the same level of abstraction is provided to them in the code structure and comments.

Many maintainers suggested that the debug interface of the code be improved. Because attitude ground support software is highly computational, an extensive debug interface is provided with each system. The problem with the current debug interface is that frequently it assumes intimate familiarity with the code in that the output was of the form ⟨**variable**⟩ = ⟨**value**⟩. Maintainers suggested that future debug interfaces provide a more descriptive explanation of the output printed.

As we learn more about the problems maintainers have with the software delivered from development and identify solutions to these problems, the guidelines and standards for development [7-9] will be modified to reflect these recommendations.

5. SUMMARY AND CONCLUSIONS

In this section, we summarize the benefits of the maintenance measurement study for the SEL, outline future maintenance measurement directions within SEL, and package some of the general lessons learned about establishing measurement programs for use in other maintenance environments.

5.1 SEL Maintenance Study Benefits

The most immediate benefit of this program has been an enhanced understanding of the SEL maintenance environment. The quantitative baselines presented in the preceding section resulted in a better understanding of maintenance requests, maintained products, and maintenance processes. They enabled us to identify weaknesses in the SEL maintenance environment.

The comparison between changes performed during development and maintenance has helped us understand where we may benefit from existing development baselines. For example, whereas the distributions of faults corrected during development and maintenance are similar, effort distributions are not. This suggests that reuse of lessons learned from development is more justified when they pertain to faults than when they pertain to effort.

Baselines may also be used to compare the effects of new development technologies on maintenance. For example, both cleanroom and an Ada/object-oriented design approach have been applied on recent development projects with the expectation that "more reliable" systems will result. We are now in a position to validate these expectations by comparing the effects of the new approaches to traditionally run projects.

In the long term, development and maintenance are expected to improve as a result of our increased understanding. At this point, recommendations for improvement are based predominantly on qualitative feedback from maintainers (rather than quantitative measurement baselines). Most of these suggestions have to do with the separation of the development and analysis organizations (Figure 1) and the absence of standard maintenance processes. The separation of development and maintenance means that a maintainer is entirely dependent on the code and documentation acquired at the time of delivery [29]. Consequently, inadequacies in the code or documentation are much more of an obstacle to maintenance than in an organization where maintenance and development are more closely related. Each maintenance change is performed by one individual without much guidance regarding the maintenance process itself. The ad hoc nature of the maintenance processes makes it hard to measure, compare measurements, and make recommendations. We expect our measurement program to contribute to the standardization of maintenance processes over time.

Overall, the SEL maintenance measurement program is perceived as successful and beneficial to this particular environment. The lessons learned from our study have resulted in changes and additions to the SEL standards and policies for software development [8]. Because numerous new projects are always under development in the SEL, we will be able to examine whether the revised standards have a measurable impact on the quality of the development product.

5.2 Future Maintenance Research

As we continue to learn about the SEL maintenance environment, numerous future measurement directions become evident. Some directions reflect changes in the environment itself, others reflect changes in our understanding of the environment. We must continually revise our goals, questions, metrics, and procedures to reflect the current priorities and understanding. Figure 12 contains an example set of revised questions for each of our seven maintenance goals to guide future maintenance studies.

We must continue to revise our measurement program in response to previous misconceptions inherent in our initial qualitative models of maintenance process. For example, our current effort classification scheme does not explicitly recognize configuration management as a discrete activity. This effort is grouped together with nontechnical activities such as meetings and management. In the future, we may want to update our data collection forms to include configuration management as a separate activity, since it seems to represent a significant portion of current maintenance effort.

GOAL 1: Characterize the changes performed during maintenance.
  QUESTION 1
    How many changes of each type are requested by different sources (e.g., analyst, operator)?
GOAL 2: Characterize product evolution during maintenance.
  QUESTION 2
    How does coupling/cohesion change during maintenance?
GOAL 3: Characterize the maintenance process stability.
  QUESTION 3
    Which process factors determined process stability (e.g., staffing level, familiarity with system)?
GOAL 4: Compare changes made during development and maintenance.
  QUESTION 4
    What is the average change effort per module during each phase?
GOAL 5: Compare changes made to products at both phases.
  QUESTION 5
    What are the distributions of requirements changes by type?
GOAL 6: Compare development and maintenance processes.
  QUESTION 6
    What are the distributions of change effort by activity.
GOAL 7: Characterize the impact of the delivered product on maintenance.
  QUESTION 7
    What product characteristics resulting from reuse have positive/negative effects on maintenance?

Figure 12. Revised measurement questions for future maintenance improvement cycles.

When our empirical investigations identify important phenomena, we must refocus our measurement goals and questions in order to study the phenomena. For example, one hypothesized implication of the stable architecture of the maintained systems (very few modules are being added or deleted) is that module cohesion within these systems may be deteriorating. Such deterioration may lead to weaker and weaker system architecture, and ultimately lead to even more difficult maintenance. Such a hypothesis needs much closer investigation before it can be presented as a potential problem.

When measurement does identify specific problems, the next step is to analyze the problems and attempt to identify viable solutions. For instance, we have quantified the types and kinds of faults uncovered during maintenance. Next, we might begin to analyze their causes in development. Such analysis may lead us to mechanisms for preventing faults, or it may help us identify better ways of detecting them.

Finally, the maintenance environment itself is continually changing. Transitions to the use of Ada and Cleanroom development in the SEL will require periodic adjustments to our measurement procedures. Such changes are not unexpected; in fact, measurement by nature must continue to evolve as the environment evolves.

5.3 Measurement Lessons Learned

The extension of the SEL into maintenance not only enabled us to gain experience with maintenance but also with establishing a maintenance measurement program [25].

Our first lesson is that there is a distinction, at least conceptually, between start-up and routine phases of measurement. During the start-up phase, there is considerable freedom to reevaluate measurement goals and redesign the metrics and procedures as our understanding of the local priorities and what is feasible grows. Once data collection forms have been designed and reflected in the data base and once people have been instructed in the procedures, it becomes expensive to introduce further changes. It is therefore critical that the start-up phase proceed cautiously. We suggest validating all measurement procedures through pilot studies.

Our second lesson concerns which questions are suitable for routine measurement. It may be tempting to use routine measurement as a mechanism for answering questions that could be resolved more efficiently by other means. For example, if the software design documentation is never maintained, it would be wasteful to discover this via routine data collection. Routine mea-

surement is appropriate for monitoring large-scale and historical trends, but it is not needed to ascertain simple facts. Many of the questions we would like to pursue are risky, i.e., we cannot be sure that the resulting data will prove useful.

Third, we have found the establishment of a measurement program in a new environment to be a time-consuming and sensitive task. Getting the program started requires building initial models of the maintenance organization, the maintained products, the maintenance processes, and the specific maintenance problems at hand. These models are used to design the measurement procedures, but must be validated during the start-up phase. Special care must also be taken to establish the creditability of measurement and win the cooperation needed to make the program a success. To collect valid data, the people providing most of the data need to be well motivated and instructed. Motivation requires addressing measurement goals of direct interest to the people providing cooperation and an opportunity for these people to review and comment on the resulting data and analyses.

Our analysis results demonstrate the immediate returns possible from investment in a measurement program. A measurement program provides invaluable insight into the processes and products within the given environment. As long as measurement is performed within a context of well defined goals and questions, such a program can be a success for any software organization.

ACKNOWLEDGMENTS

Research for this study was supported by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

We thank the CSC and GSFC personnel who have participated in our maintenance measurement program—the project personnel, the SEL data librarians, and those who reviewed earlier versions of this paper—for their tremendous support. We also thank Bruce Blum, the editor in charge of our paper, and the anonymous referees for their excellent suggestions.

REFERENCES

1. V. R. Basili, Software development: a paradigm for the future, in *Proceedings of the 13th Annual International Computer Software and Applications Conference*, 1989, pp. 471-485.
2. M. Buhler and J. Valett, Annotated Bibliography of Software Engineering Laboratory Literature, SEL-82-906, NASA/GSFC, Greenbelt, Maryland, 1990.
3. V. R. Basili, Measuring the software process and product: lessons learned in the SEL, in *Proceedings of the 10th Annual Software Engineering Workshop*, 1985.
4. F. E. McGarry, Studies and experiments in the SEL, in *Proceedings of the 10th Annual Software Engineering Workshop*, 1985.
5. R. W. Selby, Jr., and V. R. Basili, Comparing the Effectiveness of Software Testing Strategies, *IEEE Trans. Software Eng.* 13:1278-1296 (1987).
6. J. D. Valett and F. E. McGarry, A summary of software measurement experience in the Software Engineering Laboratory, in *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, 1988.
7. F. McGarry, G. Page, S. Eslinger, V. Church, and P. Merwarth, Recommended Approach to Software Development, SEL-81-205, NASA/GSFC, Greenbelt, Maryland, 1983.
8. Manager's Handbook for Software Development, rev. 1, SEL-84-101, NASA/GSFC, Greenbelt, Maryland, 1990.
9. R. Wood and E. Edwards, Programmer's Handbook for Flight Dynamics Software Development, SEL-86-001, NASA/GSFC, Greenbelt Maryland, 1986.
10. V. R. Basili and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, *IEEE Trans. Software Eng.* SE-10, 728-738 (1984)
11. V. R. Basili and H. D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Trans. Software Eng.* 758-773 (1988).
12. R. B. Grady, Measuring and Managing Software Maintenance, *IEEE Software* 4, 35-45 (1987).
13. R. Arnold and D. Parker, The dimensions of healthy maintenance, in *6th International Conference on Software Engineering*, 1982, pp. 10-27.
14. L. Belady and M. Lehman, A Model of Large Program Development, *IBM Syst. J.* 3, 225-252 (1976).
15. H. D. Rombach, A Controlled Experiment on the Impact of Software Structure on Maintainability, *IEEE Trans. Software Eng.* SE-12, 344-354 (1987).
16. C. K. S. Chong Hok Yuen, An empirical approach to the study of errors in large software under maintenance, in *Conference on Software Maintenance – 1985*, 1985, pp. 96-105.
17. H. D. Rombach and V. R. Basili, Quantitative assessment of maintenance: an industrial case study, in *IEEE Conference on Software Maintenance – 1987*, 1987, pp. 134-144.
18. B. W. Boehm and P. N. Papaccio, Understanding and Controlling Software Costs, *IEEE Trans. Software Eng.* SE-14, 1462-1477 (1988).
19. D. P. Hale and D. A. Haworth, Software maintenance: a profile of past empirical research, in *Conference on Software Maintenance – 1988*, 1988, pp. 236-240.
20. G. Heller, Data Collection Procedures for the Rehosted SEL Database, SEL-87-008, NASA/GSFC, Greenbelt, Maryland, 1987.
21. M. So, SEL Database Organization and User's Guide, SEL-89-001, NASA/GSFC, Greenbelt, Maryland, 1989.
22. H. D. Mills, Software Development, *IEEE Trans. Software Eng.* 2:265-273 (1976).
23. C. Brophy, Lessons Learned in the Transition to Ada from FORTRAN at NASA/Goddard, SEL-89-005, NASA/GSFC, Greenbelt, Maryland, 1989.

24. S. Green, A. Kouchakdjian, V. Basili, and D. Weidow, The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis, SEL-90-002, NASA/GSFC, Greenbelt, Maryland, 1990.
25. H. D. Rombach and B. T. Ulery, Establishing a measurement based maintenance improvement program: lessons learned in the SEL, *Conference on Software Maintenance – 1989*, 1989, pp. 50-57.
26. E. B. Swanson, The dimensions of software maintenance, in *Proceedings of the 2nd IEEE International Conference on Software Engineering*, 1976, pp. 492-497.
27. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
28. N. Chapin, Software Maintenance Life Cycle, in *Conference of Software Maintenance – 1988*, 1988, pp. 6-13.
29. E. B. Swanson and C. M. Beath, Departmentalization in Software Development and Maintenance, *Commun. ACM* 33, 658-667 (1990).

APPENDIX A: Data Collection Forms

WEEKLY MAINTENANCE EFFORT FORM

Name: _____

Project: _____

Friday Date: _____

For Librarian's Use Only

Number: _____

Date: _____

Entered by: _____

Checked by: _____

Section A – Total Hours Spent on Maintenance (Includes time spent on all maintenance activities for the project excluding writing specification modifications)

Section B – Hours By Class of Maintenance (Total of hours in Section B should equal total hours in Section A)

Class	Definition	Hours
Correction	Hours spent on all maintenance associated with a system failure.	
Enhancement	Hours spent on all maintenance associated with modifying the system due to a requirements change. Includes adding, deleting, or modifying system features as a result of a requirements change.	
Adaptation	Hours spent on all maintenance associated with modifying a system to adapt to a change in hardware, system software, or environmental characteristics.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

Section C – Hours By Maintenance Activity (Total of hours in Section C should equal total hours in Section A)

Activity	Activity Definitions	Hours
Isolation	Hours spent understanding the failure or request for enhancement or adaptation.	
Change Design	Hours spent actually redesigning the system based on an understanding of the necessary change.	
Implementation	Hours spent changing the system to complete the necessary change. This includes changing not only the code, but the associated documentation.	
Unit Test/ System Test	Hours spent testing the changed or added components. Includes hours spent testing the integration of the components.	
Acceptance/ Benchmark Test	Hours spent acceptance testing or benchmark testing the modified system.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

5150G(1):4

MAY 1989

Figure A1. Weekly Maintenance Report Forms.

MAINTENANCE CHANGE REPORT FORM

For Librarian's Use Only
Number: _____
Date: _____
Entered by: _____
Checked by: _____

Name: _____ OSMR Number: _____

Project: _____ Date: _____

SECTION A: Change Request Information

Functional Description of Change: _____

What was the type of modification?

- ___ Correction
- ___ Enhancement
- ___ Adaptation

What caused the change?

- ___ Requirements/specifications
- ___ Software design
- ___ Code
- ___ Previous change
- ___ Other

SECTION B: Change Implementation Information

Components Changed/Added/Deleted: _____

	< 1hr	1 hr to 1 day	1 day to 1 week	1 week to 1 month	> 1 month
Estimate effort spent isolating/determining the change:					
Estimate effort to design, implement, and test the change:					

Check all changed objects:

- ___ Requirements/Specifications Document
- ___ Design Document
- ___ Code
- ___ System Description
- ___ User's Guide
- ___ Other

If code changed, characterize the change (check most applicable)

- ___ Initialization
- ___ Logic/control structure (e.g., changed flow of control)
- ___ Interface (internal) (module to module communication)
- ___ Interface (external) (module to external communication)
- ___ Data (value or structure) (e.g., variable or value changed)
- ___ Computational (e.g., change of math expression)
- ___ Other (none of the above apply)

Estimate the number of lines of code (including comments): ____ added ____ changed ____ deleted

Enter the number of components: ____ added ____ changed ____ deleted

Enter the number of the added components that are ____ totally new ____ totally reused ____ reused with modifications

5150G(1)-3

MAY 1989

Figure A2. Maintenance Change Report Form.

SECTION 6 – ADA TECHNOLOGY

SECTION 6—ADA TECHNOLOGY

The technical papers included in this section were originally prepared as indicated below.

- "Object-Oriented Programming with Mixins in Ada," E. Seidewitz, *Ada Letters,* March/April 1992
- "Software Engineering Laboratory Ada Performance Study—Results and Implications," E. W. Booth and M. E. Stark, *Proceedings of the Fourth Annual NASA Ada User's Symposium,* April 1992

OBJECT-ORIENTED PROGRAMMING WITH MIXINS IN ADA

S10-61
136140

Ed Seidewitz
Goddard Space Flight Center
Code 552.2
Greenbelt MD 20771
(301)286-7631
eseidewitz@gsfcmail.nasa.gov

N98-17171

INTRODUCTION

My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it. And no one will know just what it is.

[Rentsch 82]

Recently, I wrote a paper discussing the lack of "true" object-oriented programming language features in Ada 83, why one might desire them in Ada and how they might be added in Ada 9X [Seidewitz 91]. The approach I took in this paper was to build the new object-oriented features of Ada 9X as much as possible on the basic constructs and philosophy of Ada 83. The object-oriented features proposed for Ada 9X [Ada9X 91b], while different in detail, are based on the same kind of approach.

Further consideration of this approach led me on a long reflection on the nature of object-oriented programming and its application to Ada. The results of this reflection, presented in this paper, show how a fairly natural object-oriented style can indeed be developed even in Ada 83. The exercise of developing this style is useful for at least three reasons:

1. It provides a useful style for programming object-oriented applications in Ada 83 until new features become available with Ada 9X;
2. It demystifies many of the mechanisms that seem to be "magic" in most object-oriented programming languages by making them explicit; and
3. It points out areas that are and are not in need of change in Ada 83 to make object-oriented programming more natural in Ada 9X.

In the next four sections I will address in turn the issues of object-oriented classes, mixins, self-reference and supertyping. The presentation is through a sequence of examples, similar to those in [Seidewitz 91]. This results in some overlap with that paper, but all the examples in the present paper are written entirely in Ada 83. I will return to considerations for Ada 9X in the last section of the paper.

CLASSES

An object *represents a component of...[a] software system... An object consists of some private memory and a set of operations...A crucial property of an object is that its private memory can be manipulated only by its operations...A* class *describes the implementation of a set of objects that all represent the same kind of system component.*

[Goldberg and Robson 83]

In Ada, an object is a variable or a constant that contains a value. The declared type of the object determines the set of possible values for the object and the set of operations that may be applied to the object. If this type is a private type, then the value of the object may only be changed through application of an operation. This corresponds to the object-oriented notion of a *class*.

Consider, for example, a simple financial account class implemented as a private type:

```
with Finance_Types; use Finance_Types;
package Finance is

  type ACCOUNT is limited private;

  procedure Open        (The_Account   : in out ACCOUNT;
                         With_Balance  : in     MONEY);
```

```
  procedure Deposit     (Into_Account  : in out ACCOUNT;
                         The_Amount    : in     MONEY);

  procedure Withdraw    (From_Account  : in out ACCOUNT;
                         The_Amount    : in     MONEY);

  function  Balance_Of  (The_Account   : ACCOUNT) return MONEY;

private

  type ACCOUNT is
    record
      Balance : MONEY := 0.00;
    end record;

end Finance;
```

The type `Finance.ACCOUNT` represents a class of account objects. The subprograms defined in package `Finance` are the allowable operations on objects of this class. The body of this package is straightforward. Note that for simplicity I will assume that a number of simple types (such as `MONEY`) are defined in a `Finance_Types` package.

The class defined by package `Finance` provides a simple but very general abstraction. In an object-oriented approach, such general classes are used as the basis for implementing more specialized classes. For example, a savings account is a specific kind of account that holds savings that earn interest. Other than some new operations associated with earning interest, a savings account is the same as the original general financial account. Thus we should be able to implement a savings account in terms of a general account:

```
with Finance_Types; use Finance_Types;
package Savings is

  type ACCOUNT is limited private;

  procedure Open           (The_Account   : in out ACCOUNT;
                            With_Balance  : in     MONEY);

  procedure Set_Rate       (Of_Account    : in out ACCOUNT;
                            To_Rate       : in     RATE);

  procedure Deposit        (Into_Account  : in out ACCOUNT;
                            The_Amount    : in     MONEY);

  procedure Withdraw       (From_Account  : in out ACCOUNT;
                            The_Amount    : in     MONEY);

  procedure Earn_Interest  (On_Account    : in out ACCOUNT;
                            Over_Time     : in     INTERVAL);

  function  Balance_Of     (The_Account   : ACCOUNT) return MONEY;

  function  Interest_On    (The_Account   : ACCOUNT) return MONEY;

private

  type ACCOUNT is
    record
      Parent       : Finance.ACCOUNT;
      Rate         : RATE := 0.06;
      Interest     : MONEY := 0.00;
    end record;

end Savings;
```

While this may not seem to gain us a lot in this simple example, such incremental extension of abstractions is fundamental to object-oriented techniques. The class of financial accounts is said to be the *superclass* of the class of savings accounts. Each savings account (of type `Savings.ACCOUNT`) has a unique *parent* financial account (of type `Finance.ACCOUNT`).

Now, three of the seven savings account operations (`Open`, `Deposit` and `Withdraw`) are syntactically and semantically the same as the corresponding financial account operations. Thus, we would like to *inherit* these financial account operations. Ada 83 has no direct way of doing this. Nevertheless, we can achieve the effect of inheritance for our present purposes by using *call-through* subprograms. For example, the `Savings.Deposit`

operation can easily be implemented as follows:

```
procedure Deposit (Into_Account : in out ACCOUNT;
                   The_Amount   : in     MONEY) is
begin
  Finance.Deposit (Into_Account.Parent, The_Amount)
end Deposit;
```

The expense of such call-throughs may be minimized through the use of pragma `Inline`.

Three other savings account operations (`Set_Rate`, `Earn_Interest` and `Interest_On`) provide the incremental new functionality of the savings account *subclass*. These operations are implemented in terms of the additional components of the representation of type `Savings.ACCOUNT`. For example:

```
procedure Earn_Interest (On_Account : in out ACCOUNT;
                         Over_Time  : in     INTERVAL) is

  Balance : constant MONEY := Balance_Of (On_Account);

begin

  if Balance > 0.00 then
    On_Account.Interest := On_Account.Interest
                         + Balance*On_Account.Rate*Over_Time;
  end if;

end Earn_Interest;
```

Note that the `Balance_Of` operation used here is the subclass operation `Savings.Balance_Of`.

The remaining savings account operation, `Balance_Of`, is syntactically the same as the financial account operation, but it is semantically different. The balance of a savings account includes interest earned up to the present point in time:

```
function Balance_Of (The_Account : ACCOUNT) return MONEY is
begin

  return Finance.Balance_Of (The_Account.Parent)
       + The_Account.Interest;

end Balance_Of;
```

Note that while `Balance_Of` is not a call-through operation, the superclass operation `Finance.Balance_Of` is used in its implementation.

The usefulness of a superclass like the financial account class comes from the fact that it can provide a common basis for a *number* of subclasses. For example, a class of checking accounts may provide another subclass of financial accounts:

```
with Finance;
with Finance_Types; use Finance_Types;
package Checking is

  type ACCOUNT is limited private;

  procedure Open       (The_Account  : in out ACCOUNT;
                        With_Balance : in     MONEY);

  procedure Set_Fee    (Of_Account   : in out ACCOUNT;
                        To_Fee       : in     MONEY);

  procedure Deposit    (Into_Account : in out ACCOUNT;
                        The_Amount   : in     MONEY);

  procedure Withdraw   (From_Account : in out ACCOUNT;
                        The_Amount   : in     MONEY);

  function  Balance_Of (The_Account  : ACCOUNT) return MONEY;

private
```

```
  type ACCOUNT is
    record
      Parent           : Finance.ACCOUNT;
      Overdraft_Fee    : MONEY := 10.00;
    end record;

end Checking;
```

In this simple example, the only difference between checking accounts and financial accounts is that overdrawing a checking account is not permitted. Further, each overdraft attempt (i.e, a returned check) is penalized by deducting a fee from the account. Thus, the implementation of `Withdraw` must be changed for checking accounts:

```
procedure Withdraw (From_Account  : in out ACCOUNT;
                    The_Amount    : in     MONEY) is
begin

  if The_Amount <= Balance_Of(From_Account) then
    Finance.Withdraw (From_Account.Parent, The_Amount);
  else
    Finance.Withdraw (From_Account.Parent,
                      From_Account.Overdraft_Fee);
  end if;

end Withdraw;
```

The savings account and checking account subclasses of the financial account class may themselves act as superclasses for even more specialized classes. Thus, a general class may be the root of a quite extended class hierarchy. Each subclass in the hierarchy incrementally extends the capabilities of its superclass, while inheriting common functionality.

MIXINS

> *A* mixin *is...a subclass definition that may be applied to different superclasses to create a related family of modified classes.*
>
> [Bracha and Cook 90]

A superclass may be used as the base for many subclasses. However, as described so far, a subclass is tied to one superclass. For instance, savings accounts are based specifically on the class defined by package `Finance`. There may be other types of accounts to which we want to added interest-bearing functionality such as that defined for savings accounts. For example, an interest-bearing checking account is basically a checking account with interest-bearing functionality added to it (or, alternatively, a savings account with checking functionality added).

Rather than recoding essentially the same interest-bearing functionality each time it is needed, we can capture this functionality in a generic package that takes a specific superclass as a parameter:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with function Balance_Of (The_Account : SUPERCLASS) return MONEY is <>;

package Interest is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent       : SUPERCLASS;
      Extension    : MIXIN;
    end record;

  procedure Set_Rate      (Of_Account  : in out ACCOUNT;
                           To_Rate     : in     RATE);

  procedure Earn_Interest (On_Account  : in out ACCOUNT;
                           Over_Time   : in     INTERVAL);

  function  Balance_Of    (The_Account : ACCOUNT) return MONEY;
```

```
  function  Interest_On   (The_Account : ACCOUNT) return MONEY;

private

  type MIXIN is
    record
      Rate          : RATE := 0.06;
      Interest      : MONEY := 0.00;
    end record;

end Interest;
```

A generic package such as this is called a *mixin* because it provides an increment of functionality which may be "mixed-into" any superclass that has the operations required to fill in the generic parameters. Typically, mixins are used within a framework of *multiple inheritance*. For example, we can reconstruct the savings account class by inheriting from *both* the financial account class and an appropriate instantiation of the interest mixin:

```
with Finance, Interest;
with Finance_Types; use Finance_Types;
package Savings is

  type ACCOUNT is limited private;

  procedure Open          (The_Account   : in out ACCOUNT;
                           With_Balance  : in     MONEY);

  ...

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Savings_Interest is
    new Interest (Finance.ACCOUNT, Finance.Balance_Of);

  type ACCOUNT is new Savings_Interest.ACCOUNT;

end Savings;
```

The `Parent` component of the `ACCOUNT` type defined in mixin `Interest` is used to inherit from the parent superclass `Finance` via a call-through. For example:

```
procedure Open (The_Account  : in out ACCOUNT;
                With_Balance : in     MONEY) is
begin
  Finance.Open (The_Account.Parent, With_Balance);
end Open;
```

The record type `Savings_Interest.ACCOUNT` is defined as a visible, rather than a private, type in the mixin to allow access to the `Parent` component. Note that it would not be possible to replace this use of a visible record component with a function that returns the parent object, because we need to use the parent as an **in out** parameter. The type `MIXIN` is never used itself outside of the mixin package.

Call-through subprograms are also needed to inherit from the mixin instantiation `Savings_Interest`. This is because the equivalent derived subprograms obtained from the derived type definition of `Savings.ACCOUNT` are hidden by the operations declared in the package specification, and in Ada 83 there must be a full subprogram body for each of these declarations. For example:

```
function  Interest_On (The_Account : ACCOUNT) return MONEY is
begin
  return Savings_Interest.Interest_On
           (Savings_Interest.ACCOUNT(The_Account));
end Interest_On;
```

Having introduced the concept of mixins, we can, of course, also create a mixin that embodies the overdraft functionality of the checking account class:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with procedure Withdraw  (From_Account : in out SUPERCLASS;
                            The_Amount   : in     MONEY) is <>;

  with function Balance_Of (The_Account  : SUPERCLASS) return MONEY is <>;

package Draft is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent          : SUPERCLASS;
      Extension       : MIXIN;
    end record;

  procedure Set_Fee  (Of_Account   : in out ACCOUNT;
                      To_Fee       : in     MONEY);

  procedure Withdraw (From_Account : in out ACCOUNT;
                      The_Amount   : in     MONEY);

private

  type MIXIN is
    record
      Overdraft_Fee  : MONEY := 10.00;
    end record;

end Draft;
```

Even our original financial account class can be converted to a mixin:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

package Monetary is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent       : SUPERCLASS;
      Extension    : MIXIN;
    end record;

  procedure Open        (The_Account   : in out ACCOUNT;
                         With_Balance  : in     MONEY);

  ...

  function  Balance_Of  (The_Account   : ACCOUNT) return MONEY;

private

  type MIXIN is
    record
      Balance      : MONEY := 0.00;
    end record;

end Monetary;
```

Of course, this mixin does not require any superclass functionality to implement its operations. However, use of the mixin construct allows monetary account functionality to be mixed into any class.

The use of mixins causes traditional class hierarchies to collapse into pieces. Each piece is a mixin that provides a well-defined increment of functionality. We can then form specific classes from these pieces by instantiating a number of mixins and inheriting all necessary functionality from them. To provide a definite starting

point for this process, we can define a *root* class that basically does nothing more than provide an empty record to which we can add mixins:

```
package Root is

  type CLASS is limited private;

private

  type CLASS is
    record
      null;
    end record;

end Root;
```

While this root class seems a bit pointless, the concept will prove useful in the next section.

At last we are ready to construct an interest-bearing checking account class without rewriting any savings account or checking account functionality. To do this, we simply mix together interest, draft and monetary account functionality. All `Interest_Bearing_Checking.ACCOUNT` operations are implemented as call-throughs to various mixin operations. Thus, from the three mixins `Monetary`, `Interest` and `Draft`, we can easily construct an interest-bearing checking account class, as well as reconstructing our original financial, savings and checking account classes.

Of course, in the actual `Interest_Bearing_Checking` package, the three mixin generics must be instantiated in a specific sequential order. In the present case, we must first establish the basic monetary account functionality, then mix in interest and draft functionality. This results in the following implementation:

```
with Root, Monetary, Interest, Draft;
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open          (The_Account   : in out ACCOUNT;
                           With_Balance  : in     MONEY);

  ...

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Checking_Finance is          -- Basic financial account
    new Monetary (Root.Class);

  package Checking_Interest is         -- Mix in interest functionality
    new Interest (Checking_Finance.ACCOUNT, Checking_Finance.Balance_Of);

  procedure Withdraw (From_Account     : in out Checking_Interest.ACCOUNT;
                      The_Amount       : in     MONEY);
  -- call-through to Finance.Withdraw

  package Checking_Draft is            -- Mix in overdraft fee functionality
    new Draft (Checking_Interest.ACCOUNT,
               Withdraw,
               Checking_Interest.Balance_Of);

  type ACCOUNT is new Checking_Draft.ACCOUNT;
                                       -- Private type representation

end Interest_Bearing_Checking;
```

Note that all the mixins are instantiated in the private part of the specification. Each instantiation uses the type and subprograms from the previous instantiation as arguments. The intermediate procedure `Withdraw` for type `Checking_Interest.ACCOUNT` is necessary because the instantiated mixin `Checking_Interest` only provides the interest-related operations on `Checking_Interest.ACCOUNT`. It is implemented as simply a call-through to `Finance.Withdraw`.

When an object of a given class is created its state components include those of the class and all its superclasses and it can perform operations of the class and its superclasses on the component state. References to "self" in operations of a superclass refer to the composite object on behalf of which the operation is to be performed.

[Wegner 87]

In the interest-bearing checking account package at the end of the last section, the `Interest` mixin was instantiated before the `Draft` mixin. It would seem that we could equally well have instantiated them in the opposite order:

```
with Root, Monetary, Interest, Draft;
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open          (The_Account   : in out ACCOUNT;
                           With_Balance  : in     MONEY);

  --

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Checking_Finance is           -- Basic financial account
    new Monetary (Root.Class);

  package Checking_Draft is             -- Mix in overdraft fee functionality
    new Draft (Checking_Finance.ACCOUNT, Checking_Finance.Withdraw,
               Checking_Finance.Balance_Of);

  function Balance_Of (The_Account : in Checking_Draft.ACCOUNT)
    return MONEY;
  -- call-through to Finance.Balance_Of

  package Checking_Interest is          -- Mix in interest functionality
    new Interest (Checking_Draft.ACCOUNT, Balance_Of);

  type ACCOUNT is new Checking_Interest.ACCOUNT;
                                        -- Private type representation

end Interest_Bearing_Checking;
```

Unfortunately, it turns out that this introduces a subtle error, as follows:

- In the new implementation, the `Draft` mixin is instantiated before the `Interest` mixin, using the `Checking_Finance.Balance_Of` operation.
- The implementation of the `Withdraw` operation in the `Draft` mixin uses the `Balance_Of` operation given as a generic formal superclass operation to determine if there is an overdraft. In this case, the actual subprogram used is `Checking_Finance.Balance_Of`, which does not add in any earned interest.
- The `Interest_Bearing_Checking.Withdraw` operation is inherited from the instantiation `Checking_Draft` of the `Draft` mixin, so as to include the overdraft functionality. This means that accumulated interest is ignored when checking for an overdraft. This is clearly unfair to the customer!

The problem is that we do not really want to use the *superclass* `Balance_Of` operation in the `Draft` mixin instantiation. Rather, we need to use the `Balance_Of` operation from the composite *subclass* being constructed. However, we cannot use the subclass type `Interest_Bearing_Checking.ACCOUNT` in the instantiation of the `Draft` mixin, because that type cannot be fully defined yet. Thus, we must instead be sure to instantiate the `Interest` mixin first, so that the interest-bearing functionality is mixed into the `Balance_Of` operation before `Draft` is instantiated.

Such order dependencies are at best annoying sources of potential errors. At worst, they can introduce circular dependencies that make it impossible to mix together certain mixins. To avoid this, we need a mechanism that allows mixins to call subclass operations in addition to superclass operations. Following the parameterization approach that led us to mixins in the first place, we can include a second generic formal type parameter in mixins to

represent the subclass.

For example, we want the `Draft` mixin to use the subclass `Balance_Of` operation:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with procedure Withdraw  (From_Account : in out SUPERCLASS;
                            The_Amount   : in     MONEY) is <>;

  type SUBCLASS is limited private;

  with function Balance_Of (The_Account  : SUBCLASS) return MONEY is <>;

  with function Self       (Parent       : SUPERCLASS) return SUBCLASS is <>;

package Draft is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent         : SUPERCLASS;
      Extension      : MIXIN;
    end record;

  procedure Set_Fee  (Of_Account   : in out ACCOUNT;
                      To_Fee       : in     MONEY);

  procedure Withdraw (From_Account : in out ACCOUNT;
                      The_Amount   : in     MONEY);

  function  Self     (This_Account : ACCOUNT) return SUBCLASS;

private

  type MIXIN is
    record
      Overdraft_Fee : MONEY := 10.00;
    end record;

end Draft;
```

The `Withdraw` operation for this mixin is then implemented as follows:

```
procedure Withdraw (From_Account : in out ACCOUNT;
                    The_Amount   : in     MONEY) is
begin

  if The_Amount <= Balance_Of(Self(From_Amount)) then
    Finance.Withdraw (From_Account.Parent, The_Amount);
  else
    Finance.Withdraw (From_Account.Parent,
                      From_Account.Extension.Overdraft_Fee);
  end if;

end Withdraw;
```

Note the use of the function `Self` to convert an object of type `Draft.ACCOUNT` to the appropriate object of type `SUBCLASS`. These odd little `Self` functions are the key to this approach. They allow us to use the subclass operations as required.

The question is, of course, how can we implement such a `Self` function? Strangely enough, we can implement it in terms of the superclass `Self` function given as a generic formal parameter:

```
function Self (This_Account : ACCOUNT) return SUBCLASS is
begin
  return Self (This_Account.Parent);
end Self;
```

Obviously, this passing of the buck must end someplace. It ends with the root class, which we reimplement as follows:

```
generic

  type SUBCLASS is limited private;

package Root is

  type CLASS is limited private;

  procedure Initialize (The_Object  : in out CLASS;
                        To_Self     : in     SUBCLASS);

  function  Self       (This_Object : CLASS) return SUBCLASS;

private

  type CLASS is
    record
      Self : SUBCLASS;
    end record;

end Root;
```

Thus the mystery is resolved: the `Self` functions all ultimately access a `Self` component defined in the root class.

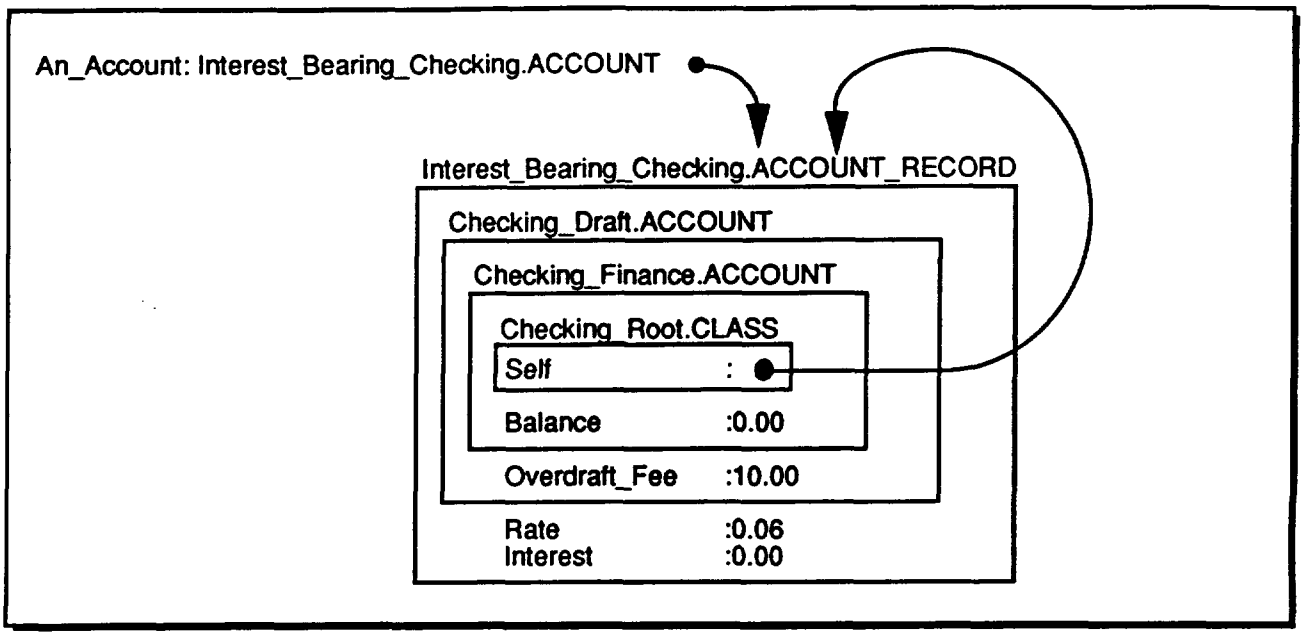
Now, the astute reader may have noticed that we have introduced a strange sort of circularity here. The representation of any class built on the root class will include a component of the subclass type. However, when we finish constructing a class from the root class and mixins, the result is the very subclass with which we need to instantiate the root class to begin with! To achieve this circularity, we must require that the subclass type be an access type. The `Self` component is then intended to be a *pointer* back to the complete, composite subclass object. (Actually, access types are also needed to allow the `Self` functions to work properly with subclass procedures that would otherwise have `in out` parameters.)

With inclusion of subclass parameters in mixins, we can now correctly implement the interest-bearing checking account class using either order of mixin instantiation:

```
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open     (The_Account  : in out ACCOUNT;
                      With_Balance : in     MONEY);

  procedure Close    (The_Account  : in out ACCOUNT);

  procedure Set_Rate (Of_Account   : in     ACCOUNT;
                      To_Rate      : in     RATE);

  ...

private

  type ACCOUNT_RECORD;
  type ACCOUNT is access ACCOUNT_RECORD;

end Interest_Bearing_Checking;
```

An advantage of implementing a private type as an access type is that the details of the type representation can be deferred to the package body by using an incomplete type definition for `ACCOUNT_RECORD` in the private part of the specification. The use of an access type also allows the use of `in` rather than `in out` parameters in procedures such as `Set_Rate`, which is necessary for the use of `Self` functions.

Circular type definition is also achieved using the incomplete type definition for `ACCOUNT_RECORD`. The circle is closed by completing the definition of `ACCOUNT_RECORD` after all the mixin instantiations in the package body. The figure on the next page shows the structure of an `Interest_Bearing_Checking.ACCOUNT` object resulting from the following implementation:




```
with Root, Monetary, Interest, Draft;
package body Interest_Bearing_Checking is

  package Checking_Root is
    new Root (SUBCLASS => Interest_Bearing_Checking.ACCOUNT);

  use Checking_Root;
  package Checking_Finance is
    new Monetary
      (SUPERCLASS    => Checking_Root.CLASS,
       SUBCLASS      => Interest_Bearing_Checking.ACCOUNT);

  use Checking_Finance;
  package Checking_Draft is
    new Draft
      (SUPERCLASS    => Checking_Finance.ACCOUNT,
       SUBCLASS      => Interest_Bearing_Checking.ACCOUNT);

  function Balance_Of (The_Account : in Checking_Draft.ACCOUNT)
    return MONEY;
  -- call-through to Finance.Balance_Of

  use Checking_Draft;
  package Checking_Interest is
    new Interest
      (SUPERCLASS    => Checking_Draft.ACCOUNT,
       SUBCLASS      => Interest_Bearing_Checking.ACCOUNT);

  type ACCOUNT_RECORD is new Checking_Interest.ACCOUNT;

  ...

end Interest_Bearing_Checking;
```

(Note that to simplify the instantiations, I have taken advantage of the box defaults on the generic formal subprogram parameters of the mixins.)

A disadvantage of using an access type is that interest-bearing checking accounts must be explicitly allocated. We can do this as part of the Open operation:

```
procedure Open (The_Account   : in out ACCOUNT;
                With_Balance  : in     MONEY) is
begin

  if The_Account /= null then
    Close (The_Account);
  end if;

  The_Account := new ACCOUNT_RECORD;
  Checking_Root.Initialize
    (The_Object => The_Account.Parent.Parent.Parent,
     To_Self    => The_Account);
  Checking_Finance.Open (The_Account.Parent.Parent, With_Balance);

end Open;
```

Note the use of the root `Initialize` operation to set the `Self` component. The figure on the previous page shows the structure of nested records and self reference that results from the allocation and initialization of an `Interest_Bearing_Checking.ACCOUNT` object.

We also need to provide a way to deallocate interest-bearing checking accounts:

```
procedure Free is new Unchecked_Deallocation (ACCOUNT_RECORD, ACCOUNT);

procedure Close (The_Account  : in out ACCOUNT) is
begin
  Free(The_Account);
end Close;
```

All the rest of the interest-bearing checking account operations are inherited from one or the other of the mixin instantiations.

SUPERTYPES

> Subtyping *is a substitutability relationship, i.e., an instance of a subtype can stand in for an instance of its supertype. How the subtype is implemented is totally irrelevant; all that matters is that it have the right behavior so that it can be substituted.*
>
> [Lalonde and Pugh 91]

Typically, the customer of a bank will have several accounts at that bank. Each bank account may be, say, a savings account, a checking account or an interest-bearing checking account. To manage all the bank accounts of one customer, we would like to create a bank account type that is the *supertype* of the types that represent the various classes of accounts. We could then create lists of bank accounts, define bank account operations, etc.

As discussed in the previous sections, each class is implemented in Ada by a private type that is distinct from all other class types. Nevertheless, we can still explicitly create a bank account supertype:

```
with Savings, Checking, Interest_Bearing_Checking;
with Finance_Types;  use Finance_Types;
package Bank is

  type ACCOUNT_TYPE is (SAVINGS, CHECKING, INTEREST_CHECKING);

  type ACCOUNT (Kind : ACCOUNT_TYPE := SAVINGS) is
    record
      case Kind is
        when SAVINGS          => A_Savings_Account   : Savings.ACCOUNT;
        when CHECKING         => A_Checking_Account  : Checking.ACCOUNT;
        when INTEREST_CHECKING => An_Interest_Checking_Account
                                  : Interest_Bearing_Checking.ACCOUNT;
      end case;
    end record;

  procedure Open       (The_Account  : in out ACCOUNT;
                        With_Balance : in     MONEY);

  procedure Close      (The_Account  : in out ACCOUNT);
```

```
   procedure Deposit    (Into_Account : in out ACCOUNT;
                         The_Amount   : in     MONEY);

   procedure Withdraw   (From_Account : in out ACCOUNT;
                         The_Amount   : in     MONEY);

   function  Balance_Of (The_Account  : ACCOUNT) return MONEY;

end Bank;
```

The type `Bank.ACCOUNT` defines a supertype with *subtypes* `Bank.ACCOUNT(SAVINGS)`, `Bank.ACCOUNT(CHECKING)` and `Bank.ACCOUNT(INTEREST_CHECKING)`. Each subtype corresponds to one of the classes defined in previous sections. Note that a private type is unnecessary here, because we wish to be able to freely convert between the `Bank.ACCOUNT` subtypes and the class types.

The five operations defined in package `Bank` reflect the operations that are common to all the account types. Semantically, we wish each supertype operation to mirror the implementation of the appropriate subtype operation. For example, the statement:

```
Bank.Withdraw (From_Account => A, The_Amount => X);
```

should be equivalent to either `Savings.Withdraw`, `Checking.Withdraw` or `Interest_Bearing_Checking.Withdraw`, depending on the subtype of `A`. Since the subtype of `A` can, in general, only be determined at run-time, we are effectively asking that `Bank.Withdraw` be *dynamically bound* to the appropriate subtype operation.

We can achieve the effect of dynamic binding in Ada by implementing the bank account operations as *dispatching* or *case-selection* subprograms. For example:

```
procedure Withdraw (From_Account  : in out ACCOUNT;
                    The_Amount    : in     MONEY) is
begin

  case Kind is
    when SAVINGS =>
      Savings.Withdraw (From_Account.A_Finance_Account, The_Amount);

    when CHECKING =>
      Checking.Withdraw (From_Account.A_Checking_Account, The_Amount);

    when INTEREST_CHECKING =>
      Interest_Bearing_Checking.Withdraw
        (From_Account.An_Interest_Checking_Account, The_Amount);

  end case;

end Withdraw;
```

Once we have the bank account supertype, we can create *polymorphic* data structures and operations that can handle all kinds of bank accounts. For example:

```
type CUSTOMER_ACCOUNTS is array(POSITIVE range <>) of Bank.ACCOUNT;

function Total_Assets_Of (The_Accounts : CUSTOMER_ACCOUNTS) return MONEY is

  Total : MONEY := 0.00;

begin

  for I in The_Accounts'range loop
    Total := Total + Bank.Balance_Of (The_Accounts(I));
  end loop;

  return Total;

end Total_Assets_Of;
```

The function defined above finds the total assets a customer has in his accounts, regardless of what kinds of accounts they are.

It is important to note that to be included in a supertype, a class need only provide implementations for all the operations defined for the supertype. The ways in which various subtype classes implement these operations do not have to be related at all. For example, the `Bank.ACCOUNT` supertype is constructed from a number of classes implemented by various combinations of the mixins `Monetary`, `Interest` and `Draft`. These classes thus share some common implementation, but this is not at all important to the construction of the supertype.

Thus, supertypes and superclasses are really distinct concepts. Looking at it another way, the supertype provides a set of dispatching operations for those operations which are common to all its subtypes, regardless of how those operations may be implemented by the subtype classes or how the subtypes may be represented. A supertype that is constructed in this way from a given list of subtype classes is said to be the *union type* of those classes. Thus we have constructed a bank account supertype that is the union of the savings, checking and interest-bearing checking account classes.

It was noted earlier that the use of mixins causes a collapse of the original class hierarchy. Using union types, however, we can still form a type hierarchy by appropriately grouping classes. As well as the `Bank.ACCOUNT` union type, such a *type* hierarchy for account classes could include the union of the savings and interest-bearing checking account classes (an investment supertype treating interest-bearing checking accounts as savings accounts) and the union of the checking and interest-bearing checking account classes (a cash account supertype treating interest-bearing checking accounts as checking accounts). Note how it is possible for a class to be included in more than one union type.

CONSIDERATIONS FOR ADA 9X

> *There is a recognized need for improving Ada's support for data abstraction, and the construction of programs from pre-existing components.*
>
> [Ada9X 91a]

The mixin-based style described in this paper combines the benefits of object-oriented mixins with the advantages of explicit parameterization through generics. With superclass and subclass parameterization, mixins are completely independent software components that can be mixed and matched in many combinations. This leads to a powerful paradigm known as parameterized programming that promotes highly reusable code (see, for example, [Goguen 84; Seidewitz and Stark 91]).

Unfortunately, as the reader can see from the examples in this paper, this style is awkward in places with Ada 83. In particular, the following areas especially need to be addressed in Ada 9X:

1. There needs to be a way to create a subclass type by simple extension of a class type and to parameterize this extension with a mixin. The proposed Ada 9X record extension mechanism [Ada9X 91b] fills this need admirably well.

2. There needs to be a simpler way to achieve self-reference during the combination of mixins. This need seems to be filled by the proposed mechanism in Ada 9X to allow type extensions as generic formal type parameters [Ada9X 91b]. This would probably necessitate the use of nested generics to allow the mixin type to be an extension of the `SUPERCLASS` type parameter and the `SUBCLASS` type parameter to be an extension of the mixin type. Such a construction would, however, eliminate the need for `Self` functions.

3. There needs to be a mechanism for constructing supertypes without having to explicitly code dispatch operations. Ada 9X does provide an automatic dispatching capability using "tagged records" [Ada9X 91b]. However, this capability can only be used if the subtypes are implemented as subclasses (type extensions) of the supertype. This perpetuates the confusion of superclass and supertype.

Thus the proposed object-oriented features for Ada 9X largely support the mixin style described in this paper. Unfortunately, the tagged record mechanism confuses type extension and dispatching. This is analogous to the equation of superclasses and supertypes in most typed object-oriented programming languages (such as C++ [Stroustrup 86]).

Requiring supertypes to be superclasses is inconvenient when we are using generic mixins to construct classes, and wish to create a type hierarchy after the fact. Perhaps a better model for Ada 9X would be the "abstract type" mechanism of the languages Emerald [Black et al. 87] and POOL-I [America and van der Linden 90]. Even with the currently proposed Ada 9X features, however, a generics-based approach to mixins, such as that presented in this paper, could be an important contribution of Ada 9X back to the object-oriented programming community.

ACKNOWLEDGEMENT

I would like to thank my colleague Mike Stark for a number of good suggestions that greatly improved the clarity of presentation of this paper.

Ada9X 91a — *DRAFT Mapping Rationale Document*, Ada 9X Project Report, February 1991

Ada9X 91b — *Ada 9X Mapping Document*, Draft Ada 9X Project Report (2 volumes), August 1991

America and van der Linden 90 — Pierre America and Frank van der Linden, "A Parallel Object-Oriented Language with Inheritance and Subtyping", *Proceedings of the Conference on Object-Oriented Programming System, Languages, and Applications / European Conference on Object-Oriented Programming, SIGPLAN Notices*, October 1990

Black et al. 87 — Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy and Larry Carter, "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, January 1987

Bracha and Cook 90 — Gilad Bracha and William Cook, "Mixin-Based Inheritance", *Proceedings of the Conference on Object-Oriented Programming System, Languages, and Applications / European Conference on Object-Oriented Programming, SIGPLAN Notices*, October 1990

Goguen 84 — Joseph A. Goguen, "Parameterized Programming", *IEEE Transactions on Software Engineering*, September 1984

Goldberg and Robson 83 — Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983

Lalonde and Pugh 91 — Wilf LaLonde and John Pugh, "Subclassing ≠ Subtyping ≠ Is–a", *Journal of Object-Oriented Programming*, January 1991

Rentsch 82 — "Object-Oriented Programming", *SIGPLAN Notices*, September 1982

Seidewitz 91 — Ed Seidewitz, "Object-Oriented Programming through Type Extension in Ada 9X", *Ada Letters*, March/April 1991

Seidewitz and Stark 91 — Ed Seidewitz and Mike Stark, "An Object-Oriented Approach to Parameterized Software in Ada", *Proceedings of the Eighth Washington Ada Symposium*, June 1991

Stroustrup 86 — Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986

Wegner 87 — Peter Wegner, "The Object-Oriented Classification Paradigm", in *Research Directions in Object-Oriented Programming*, ed. by Bruce Shriver and Peter Wegner, The MIT Press, 1987

SOFTWARE ENGINEERING LABORATORY ADA PERFORMANCE STUDY-RESULTS AND IMPLICATIONS

Eric W. Booth
Computer Sciences Corporation
Lanham-Seabrook, Maryland
(301) 794-1277

Michael E. Stark
NASA/Goddard Space Flight Center
Greenbelt, Maryland
(301) 286-5048

SUMMARY

The Ada Language Reference Manual (LRM) (Reference 1) states:

> "Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency."

Initial implementations of Ada compilers and development environments tended to favor the first two concerns over the concern for efficiency. Similarly, initial (non-real-time, non-embedded) applications development using Ada as the programming language tended to favor maintainability, readability, and reusability.

As software systems become more sophisticated the need to predict, measure, and control the run time performance of systems in the Flight Dynamics Division (FDD) is a growing concern. The transition to Ada introduces performance issues that were previously nonexistent. More-over, this transition is often accompanied by the transition to object-oriented development (OOD), which has performance implications, independent of the programming language, that must be considered. To better understand the implications of new design and implementation approaches, the Software Engineering Laboratory (SEL) conducted an Ada performance study.

The SEL is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) to investigate the effectiveness of software engineering technologies applied to the development of applications software. The SEL was created in 1977 and has three organizational members: NASA/GSFC, Systems Development Branch; The University of Maryland, Computer Sciences Department; and Computer Sciences Corporation, Systems Development Operation.

The goals of the SEL are (1) to understand the software development process in the GSFC environments; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes the *Ada Performance Study Report* (Reference 2).

This paper describes the background of Ada in the FDD, the objectives and scope of the Ada Performance Study, the measurement approach used, the performance tests performed, the major test results, and the implications for future FDD Ada development efforts.

APPROACH TO MEASUREMENT

To measure the run-time performance of design alternatives and language features, two fundamental approaches were used. The first approach measured the run-time improvement of existing systems after an alternative had been incorporated into a baseline version of the system. The second approach used the ACM SIGAda PIWG test suite and added tests specific to the flight dynamics environment.

Overview

Benchmark programs are commonly used to evaluate the performance of design alternatives and language features. Such benchmark programs include (1) sample applications such as sorting programs or, as in the FDD, simulators, (2) programs to measure the overhead associated with a design alternative or language feature, and (3) synthetic benchmarks designed to measure the time needed to execute a representative mix of statements (e.g., Whetstone, Dhrystone) (Reference 6). The first approach used by this study falls into the first benchmark category, and the second approach falls into the last two.

To measure the overhead of a design alternative or language feature, the dual-loop approach is used to subtract the overhead associated with control statements that aid in performing the measurement. This approach uses a control loop and a test loop; the test loop contains everything contained in the control loop and the alternative being measured. A major factor in designing a dual-loop benchmark is compiler optimization. It is critical that the code generated by the compiler for both loops be identical except for the quantity being measured (Reference 7). In addition, it is necessary to ensure that the statement or sequence of statements being tested does not get optimized away.

Although the dual-loop approach can be used for synthetic benchmarks and applications, this technique is not required if the run time of the program is long in comparison to the system clock resolution (Reference 7). Instead, the CPU time can be sampled at the beginning of the program and again after a number of iterations of the program. The time for the benchmark/application is then (CPU_Stop - CPU_Start)/Number_Iterations. The same measurement can be achieved by submitting the test program to run as a batch job and obtaining the CPU time from the batch log file. This CPU time can then be divided by the number of times the sequence of statements being measured is executed in the main control loop of the test program.

It is important to understand the run-time environment in which the benchmarks are run when interpreting test results. VMS checks the timer queues once per second, which can affect measurement accuracy. Under VMS, the Ada run-time system is bundled with the release of the operating system and installed as a shareable executable image. Consequently, DEC Ada performance is directly dependent on the installed version of VMS. There is also a degree of uncertainty when using CPU timers provided in time-shared systems like VMS. In the presence of other jobs, CPU timers charge ticks to the running process when the wall clock is updated. It is therefore possible for time to be charged to active processes inaccurately because context switches can occur at any time. Finally, it cannot be assumed that running benchmarks for a hosted system in batch during low usage (such as, at 11 pm) guarantees standalone conditions (References 7 and 8). Therefore, the FD benchmarks to test individual design alternatives were run on the weekend to minimize these effects.

THE FIRST APPROACH -- SIMULATOR

Several of the design alternatives examined by this study were tested and analyzed in the context of two FDD simulators. Alternatives were chosen to be implemented in the context of these simulators for the following reasons:

1. They were simulator-specific, e.g., different ways of implementing the scheduler.

2. They could be implemented in an isolated part of the simulator where their impact could easily be measured using the VAX PCA.

3. They could be implemented in an isolated part of the simulator and still have a measurable effect on the time required for a 20-minute simulation run.

Baselined versions of the simulators were used to test each of the design alternatives. CPU times were obtained for 20-minute simulation runs of the baselined versions from the log files created by batch runs. PCA was used to obtain a profile of the simulators. These profiles showed what percentage of the CPU time was spent in each Ada package of the simulator. The VAX manual *Guide to VAX Performance and Coverage Analyzer* (Reference 9) contains more information on PCA.

Design alternatives were incorporated into the baselined versions of the simulators. New CPU times were obtained for 20-minute simulation runs from the log files created by batch runs and new profiles obtained using PCA. The following two figures show the accounting information contained in a batch log file and a sample of PCA output. From these two pieces of information, the impact of each design alternative was assessed.

Sample PCA Output

```
                VAX Performance and Coverage Analyzer
            CPU Sampling Data (11219 data points total) - "**"

Bucket Name          +----+----+----+----+----+----+----+----+----+----+
PROGRAM_ADDRESS\     |
 UTILITIES_ . . . . .|**************************************************  23.2%
 SIMULATION_SCHEDULE|**********************                              10.7%
 SEARCH_STRING  . . |****************                                     7.5%
 SPACECRAFT_ATTITUDE|****************                                     7.5%
 DATABASE_MANAGER . |***************                                      7.0%
 ADDING_UTILITIES . |**********                                         4.7%
 EARTH_SENSOR . . . |********                                           3.7%
 UTILITIES_LONG_  . |******                                             3.0%
 DATABASE_TYPES_  . |******                                             2.9%
 SPACECRAFT_WHEELS  |******                                             2.9%
 AOCS_PROCESSOR . . |******                                             2.6%
 SPACECRAFT_EPHEMERI|*****                                              2.5%
 ENVIRONMENTAL_TORQU|*****                                              2.3%
 THRUSTERS  . . . . |*****                                              2.2%
 GEOMAGNETIC_FIELD  |*****                                              2.2%
 DEBUG_COLLECTOR  . |****                                               2.1%
 MAGNETOMETER . . . |****                                               1.7%
 SADA . . . . . . . |***                                                1.4%
 SOLAR_SYSTEM . . . |***                                                1.2%
                    +----+----+----+----+----+----+----+----+----+----+
```

Sample Batch Log File Accounting Information

```
Accounting information:
 Buffered I/O count:          109      Peak working set size:  4096
 Direct I/O count:           1132      Peak page file size:   15304
 Page faults:               11766      Mounted volumes:           0
 Charged CPU time:     0 00:06:45.08   Elapsed time:     0 00:09:02.47
```

THE SECOND APPROACH -- PIWG

Design alternatives not isolated to a particular part of either of the simulators were tested using the PIWG structure of measurements. The PIWG structure of measurements is based on the concept of a control loop and a test loop. The test loop contains everything in the control loop and one alternative to be measured. The CPU time is sampled before the execution of each loop and after many iterations of each loop. If the test loop time duration is not considered stable, the process is repeated with a greater number of iterations; this is accomplished through an outer loop surrounding the test and control loops. To be considered stable, the test loop time duration must be greater than a predefined minimum time. If this condition is met, the test loop time duration is compared against the control loop time duration, and the number of iterations is compared against a predefined minimum number of iterations. If the test loop time is greater than the control loop time or the minimum number of iterations has been exceeded, the results are considered stable, and the CPU time for the design alternative is calculated. The time for the alternative is the difference between the amount of CPU time taken for the control loop and the amount of CPU time taken for the test loop, divided by the total number of iterations performed. Collecting control loop and test loop CPU times, calculating design alternative times, and testing for stability were done using PIWG's Iteration package in the test drivers for this study.

All test drivers used in this study were called three times from a main driver routine so that the CPU time for a given design alternative could be averaged for more accuracy. All results were averaged and recorded using PIWG's I/O package and report generator procedure. The following is a sample PIWG report.

Sample PIWG Report

```
Test Name:   Generic_A                    Class Name:  Matrix - Gen
CPU Time:      117.2  microseconds
Wall Time:     117.2  microseconds        Iteration Count:   128
                                          Number of samples: 3

Test Description:
Use of generic matrix processing
- Generic package for 3x3 matrix


Test Name:   Generic_C                    Class Name:  Matrix - Gen
CPU Time:      117.2  microseconds
Wall Time:     117.2  microseconds        Iteration Count:   128
                                          Number of samples: 3

Test Description:
Use of generic matrix processing
- NonGeneric package for 3x3 matrix
```

TEST OVERVIEW

Ten test groups were developed. Each test group represented a design or implementation issue relevant to current FDD applications. The test groups were chosen as a result of an in-depth analysis of several PCA runs with two FDD simulators. If a certain design alternative or language feature appeared to consume a relatively large portion of central processing unit (CPU) time or memory, it was analyzed, measured, and quantified in this study. The design alternatives or language features consuming a relatively small portion of CPU time or memory were not studied further. Therefore, the test groups presented here are intended to be a representative sampling, rather than an exhaustive sampling, of current design and implementation approaches. The test groups are presented in two categories: design-oriented tests and implementation-oriented tests.

Design-Oriented Tests

Following is a brief description of the purpose of each design test group performed on the Ada performance study.

Group 1: Scheduling. This test group contained three tests that addressed the run-time cost of various scheduling alternatives. This test compared a event-driven design against a time-driven design and a hard-coded design. The event-driven design maintains a prioritized (sorted) queue of event identifiers that specifies the time-step and next simulation event. The time-driven design iterates over an array of event identifiers for each fixed time-step. The hard-coded design contains the event (procedure) calls in the source code. With the event-driven design the user may vary the order and frequency of each event. In the time-driven design the user may only vary the order of the event. In the hard-coded design there are not options available to the user. The implications of the different approaches were analyzed and contrasted. The results of this test group provided the applications designers with information necessary to make trade-off decisions among flexibility, accuracy, and performance.

Group 2: Unconstrained Structures. Leaving data structures unconstrained allows greater user flexibility and enhances future reusability. However, the additional run-time code that may be generated can impose a significant run-time and memory overhead. This group measured the expense of unconstrained records and arrays and proposed viable alternatives.

Group 3: Initialization and Elaboration. This test group addressed initialization of static and dynamic data using various combinations of elaboration-time and execution-time alternatives. This test group was relevant for applications requiring minimal initialization time.

Group 4: Generic Units. The benefits of using generic units are reduced source-code size, encapsulation, information hiding, decoupling, and increased reuse (Reference 10). However, many Ada compilers implement this language feature poorly. This test group addressed the options available with the compiler implementation and how well these options were implemented.

Group 5: Conditional Compilation. The ability to include additional "debug code" in the delivered system adds to the system size and imposes a run-time penalty even if it is never used. The test group analyzed the current approach and proposed flexible alternatives for future systems. The results of this test group can have applications beyond "debug code" elimination.

Group 6: Object-Oriented Programming. Two of the fundamental principles of object-oriented programming (OOP) are polymorphism and inheritance. Ada does not directly support these principles. However, the designer may simulate the effect of inheritance and polymorphism through the use of variant

records and enumeration types. These OOP principles, whether direct or indirect, incur certain run-time overhead and problems (Reference 11).

Implementation-Oriented Tests

Following is a brief description of the purpose of each implementation test group performed on the Ada performance study.

Group 7: Matrix Storage. The most basic, and perhaps the most common, mathematical expressions in flight dynamics applications involve matrix manipulations. This group addressed row-major versus column-major algorithms to quantify the performance implications.

Group 8: Logical Operators. The Ada LRM clearly defines the behavior of logical expression evaluation. The *Ada Style Guide* (Reference 12) recommends avoiding the short-circuit forms of logical operators for performance reasons. The implications of this recommendation in the flight dynamics environment were measured and analyzed.

Group 9: Pragma Inline. Flight dynamics simulators contain a large number of procedure and function calls to simple call-throughs and selectors. The overhead of making these calls can slow the performance of any simulator. This test measured the use of **pragma** INLINE as an alternative to calling a routine.

Group 10: String-to-Enumeration Conversion. Current flight dynamics simulators contain a central logical data base. The physical data are distributed throughout the simulator in the appropriate packages. The logical data base provides *keys* (strings) that map into the physical data. The logical data base converts these strings to the appropriate enumeration type to retrieve the corresponding data. This test assessed the performance implications of this approach.

Test Documentation

Each performance test in this report is described in this section in the following format:

Purpose. Each test was designed with a specific design or implementation alternative in mind. The rationale for the choice of the alternatives tested results from analysis of existing Ada systems developed in the FDD.

Method. Some tests were performed as changes to an existing system, while other tests were performed by creating new, special-purpose software. The basis for each method was one of two approaches: DEC's PCA measurement tool or the PIWG structure of measurements. The details of the method(s) used for each test are described.

Results. The result of executing a test is some combination of CPU time and object code size. Most tests were designed to measure the CPU run time in microseconds (μs). In some cases the object code size in bytes is relevant. The data resulting from each test run are provided.

Analysis. In many cases, detailed analysis of the test results is necessary to understand the implications for future projects. The analysis performed is summarized, and the implications are highlighted.

RESULTS

As a result of this performance study, more accurate estimation of run-time performance for future FDD simulators is possible. Assuming future dynamics simulators are similar in function to GOADA, a more accurate performance estimation is possible given the following information:

1. The run-time performance for a typical run of the GOADA simulator is 6 minutes, 45 seconds for a 20-minute simulation. This yields a 1:3 simulation time to real-time ratio.

2. The performance profile generated by PCA of a typical GOADA run shows the distribution of the CPU run time resource throughout the simulator.

3. The measured results of this study that lead to more efficient design and implementation alternatives.

The following table combines the results of the Ada Performance Study with the PCA performance profile of GOADA. Each row of the table is measured against the baseline of 6 minutes and 45 seconds of CPU time to perform a 20 minuter simulation.

Impact of Measured Performance Results on Dynamics Simulators

Alternative	GOADA	Study Results	Difference
1. Looping scheduler	10.7%	2.2%	8.5%
2. Bypass logical data base	14.5%	1.8%	12.7%
3. Conditional compile debug code	2.1%	0.0%	2.1%
4. Use static data structures	45.0%	13.0%	32.0%
5. Optimized utility packages	26.6%	5.3%	21.3%
Total Percentages	98.9%	22.3%	76.6%

The first row of the table shows the performance difference between the baseline scheduler in GOADA and the looping scheduler alternative (see test group 1, scheduling). Another option is to use the "hard-coded" approach for the scheduler. However, the hard-coded approach sacrifices all flexibility in the interest of performance. For this reason, the more flexible "looping" alternative is recommended.

The second row highlights the difference between accessing the logical data base and accessing the physical data directly (see test group 10, string-to-enumeration conversion). This striking improvement came from removing one string-to-enumeration type conversion from the main simulation loop. The third row recommends the conditionally compiled debug code (see test group 5, conditional compilation). The fourth row is the estimated result of using a static record structure instead of a dynamic structure in all simulator packages (see test group 2, unconstrained structures).

The fifth row is based on the result of comparing GOADA's baseline matrix multiply function to the optimized matrix multiply function (Reference 13). Since the FDD deals with mainly three dimensions, an optimized set of utilities can be developed on that basis. The fully optimized version required less than one-fifth of the CPU time required for the baseline version.

As this table shows, a dynamics simulator that is similar to GOADA and is implemented with the results of this study would consume 76.6 percent less CPU than the current version, more than quadrupling the speed. This would yield an upper bound estimate of 95 seconds to perform a 20-minute simulation run, or approximately a 1:13 simulation-time-to-real-time ratio.

This estimate is an upper bound for three reasons. First, this study examined a representative, rather than an exhaustive, list of design and implementation alternatives. That is, only those alternatives that held the most promise of a large performance difference were studied. There may be many other alternatives that offer only minor gains. However, the combined performance gain of all may be significant.

Second, coding optimizations to GOADA, or any simulator, were not studied. The goal of the study was to identify those design and implementation alternatives that lead to optimal systems. Line-by-line micro-optimizations on a simulator only provide information on final efficiency and lack the needed information on *how* to systematically predict and achieve that level of efficiency.

Finally, the DEC Ada 1.5-44 compiler is a relatively error-free first attempt at an Ada compilation system. The next generation of Ada compilers, which includes DEC Ada 2.0, are now available. These second-generation compilation system includes improvements to the optimizer and code-generator. For example, simply compiling GOADA using DEC Ada 2.0 improved the simulator's performance by 7.4%.

CONCLUSIONS

The following statements summarize the results of the Ada Performance Study:

- Design and implementation decisions that favored fidelity over efficiency were the largest contributor to poor run-time performance. The design should continually be reevaluated against evolving user requirements and specifications.

- Ada simulators in the FDD can be designed and implemented to achieve run times comparable to those of existing FDD FORTRAN simulators. Inefficient systems indicate problems in the system design or the compiler being used.

- Current Ada compilation systems still have inconvenient features that may contribute to poor performance. Organizations using Ada should use available performance-analysis tools to assess their compilation systems.

Design changes are much more expensive than coding changes during final system testing. Often due to schedule and budget constraints, design changes are impossible. Therefore the important implication of the Ada performance study results is that new technology (in this case Ada and OOD) requires performance prototyping and benchmarking early in the design phase even in seemingly simple or straightforward cases.

The *Ada Performance Study Report* (Reference 2) contains a detailed analysis of each alternative studied and summarizes the results of this analysis with specific performance recommendations for future OOD/Ada development efforts in the FDD. Different application domains may be able to apply these results and recommendations. However, this does not preclude the necessity for application domain specific prototyping and benchmarking to determine the application specific performance issues.

REFERENCES

1. *Ada Programming Language*, American National Standards Institute/Military Standard 1815A, 1983 (ANSI/MIL-STD-1815A-1983)

2. Goddard Space Flight Center, SEL-91-003, *Software Engineering Laboratory Ada Performance Study Report*, E. Booth and M. Stark, July 1991

3. Goddard Space Flight Center, SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby, et al., prepared by Computer Sciences Corporation, December 1988

4. Goddard Space Flight Center, FDD/552-90/010, *Software Engineering Laboratory (SEL) Study of the System and Acceptance Test Phases of Four Telemetry Simulator Projects*, D. Cover, prepared by Computer Sciences Corporation, September 1990

5 Goddard Space Flight Center, SEL-89-005, *Lessons Learned in the Transition to Ada from FORTRAN at NASA/Goddard*, C. Brophy, University of Maryland, November 1989

6. Clapp, R., and T. Mudge, *Rationale, Chapter 1 - Introduction*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, NY, ISBN 0-89791-354-x, vol. 10, no. 3, Winter 1990

7. Clapp, R., and T. Mudge, Rationale, *Chapter 3 - The Time Problem*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, NY, ISBN 0-89791-354-x, vol. 10, no. 3, Winter 1990

8. Gaumer, D. and D. Roy, *Results, Reporting Test Results*, Ada Performance Issues, Ada Letters, A Special Issue, Association of Computing Machinery, New York, NY, ISBN 0-89791-354-x, vol. 10, no. 3, Winter 1990

9. Digital Equipment Corporation, *Guide to VAX Performance and Coverage Analyzer*, June 1989

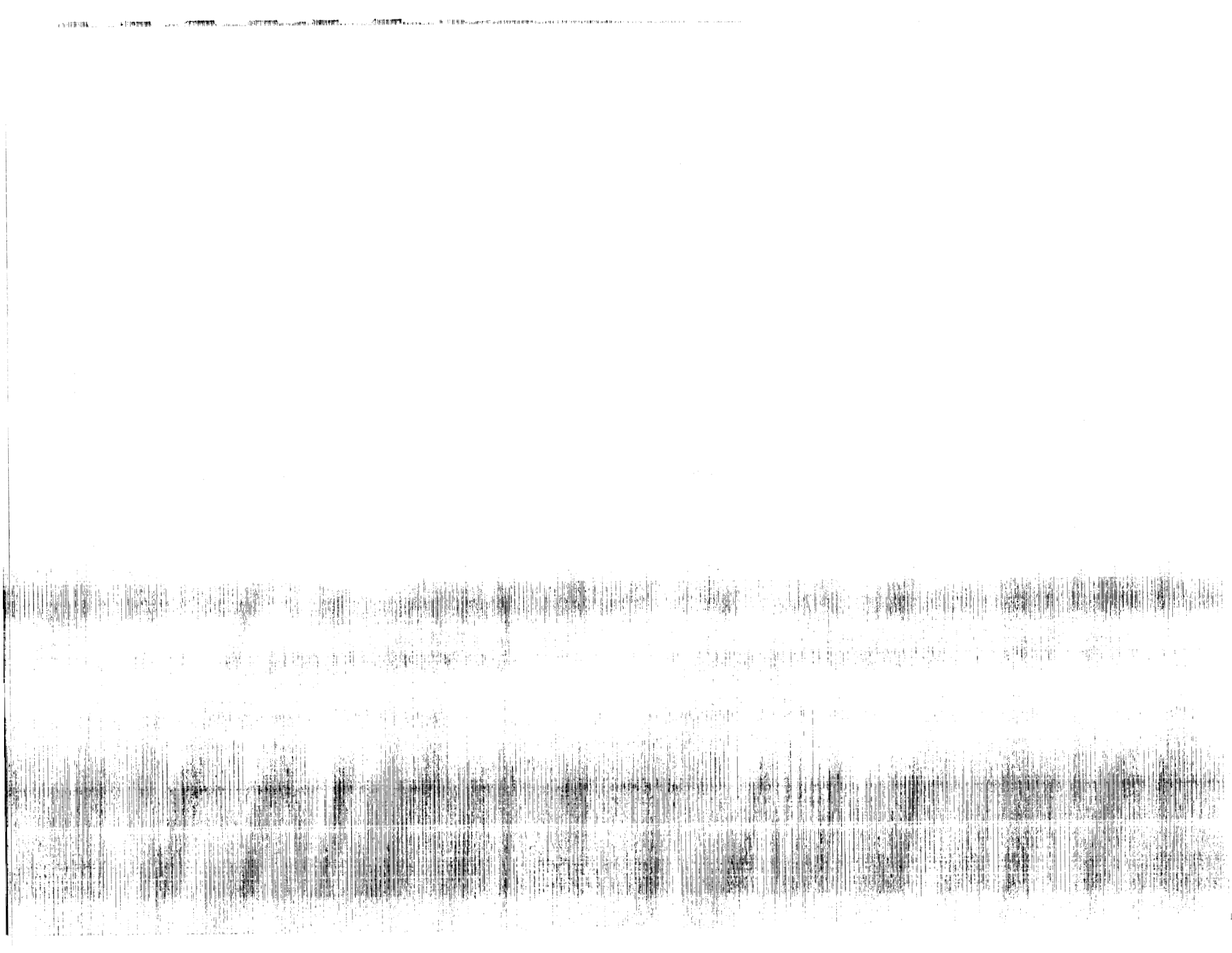
10. Goddard Space Flight Center, FDD/552-90/045, *Extreme Ultraviolet Explorer (EUVE) Telemetry Simulator (EUVETELS) Software Development History*, E. Booth and R. Luczak, prepared by Computer Sciences Corporation, 1990

11. Booch, G., *Object Oriented Design*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, ISBN 0-8053-0091-0, 1991

12. Goddard Space Flight Center, SEL-87-006, *Ada Style Guide*, E. Seidewitz et al., June 1986

13. Burley, R., "Some Data from Ada Performance Study" internal FDD memorandum, September 1990

STANDARD BIBLIOGRAPHY OF SEL LITERATURE



STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1106, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1992

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-006, *Monitoring Software Development Through Dynamic Variables*, C. W. Doerflinger, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/ Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/ Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

10000229
1118/900

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-201, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 2)*, L. Morusiewicz, J. Bristow, et al., October 1992

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

10000229
1118/900

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory," *ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V., G. Caldiera, F. McGarry, et al., "The Software Engineering Laboratory—An Operational Software Experience Factory," *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS '92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

N93-17161

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 92	Contractor Report (UM & CSC)/GSFC

4. TITLE AND SUBTITLE
Collected Software Engineering Papers: Vol. X

5. FUNDING NUMBERS
ASG5445

6. AUTHOR(S)
t

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)
NASA/GSFC Software Engineering Branch
Univ. of MD Dept. Computer Science
Computer Sciences, Software Engineering Operation

8. PEFORMING ORGANIZATION REPORT NUMBER
SEL 92 003

9. SPONSORING / MONITORING ADGENCY NAME(S) AND ADDRESS (ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING ADGENCY REPORT NUMBER
SEL 92 0

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document is a collection of technical papers produced 10/91-11/92. The purpose is to make available results of SEL research that originally appeared in different forums. These papers cover topics in software engineering, but do not encompass entire scope of SEL activities. There are 5 sections:

The Software Engineering Laboratory
Software Tools Studies
Software Models Studies
Software Measurement Studies
Ada Technology Studies

SEL is working to improve software development at GSFC. Future efforts will be documented under the same title.

14. SUBJECT TERMS
12-1/2 page bibliography

15. NUMBER OF PAGES
app. 100

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL